A MODULAR DISTRIBUTED CONSTRAINT LOGIC PROGRAMMING SYSTEM

By

ALEXEI ARUN KARVE

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KANPUF

JANUARY, 1993

A MODULAR DISTRIBUTED CONSTRAINT LOGIC PROGRAMMING SYSTEM

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

By
ALEXEI A. KARVE

to the
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
January, 1993

Dedicated

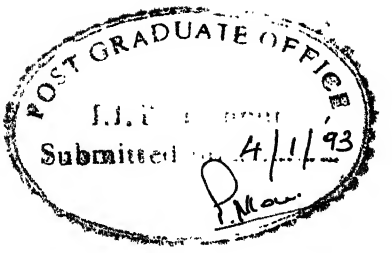
to

My Parents

Divide each problem that you examine into as many parts as you can and as you need, to solve them more easily.

Descartes.

Certificate



It is certified that the work contained in the thesis entitled *A Modular Distributed Constraint Logic Programming System*, by *Alexei A. Karve*, has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

January, 1993

Dr. Harish Karnick,
Associate Professor,
Department of Computer
Science and Engineering,
I.I.T. Kanpur - 208 016.

Dr. Sanjeev Kumar,
Assistant Professor,
Department of Computer
Science and Engineering,
I.I.T. Kanpur - 208 016.

Acknowledgements

First and foremost, I am indebted to my thesis advisors Dr. Harish Karnick and Dr. Sanjeev Kumar who have given me constant support throughout the course of the project. The thought provoking discussions have helped a lot in chanelling the work thus preventing any unnecessary veering from the main subject. Their insistence on good programming techniques and encouragement has helped in many of the examples. Their inspiring suggestions and comments have played an important role in this thesis. I would like to specially thank them for their total cooperation and timely help, the value of which cannot be measured.

I express my deep gratitude towards Dr. Sanjeev Saxena for his genial approach and guidance in the courses I undertook under him. The knowledge I have gained has helped me a lot in making decisions throughout the project.

I am grateful to Mr. Vishal Saxena for providing access to Mathematica in the Chemical Department and to Mr. S. Sen from the Robotics Department for making the package REDUCE available to me.

I thank Mr. M. R. Gaikaiwari for introducing me to Compilers. He has been a perennial source of inspiration and encouragement all through.

My sincere thanks to the staff of the Department Library, the staff of the Central Library and the technical staff of the Computer Center for providing timely assistance.

My tender thanks to my fiancèe Geeta whose thoughts compelled me in completing the thesis at the earliest.

I would like to thank all my collegues for their moral support and specially mention the names of Harshad Parekh, Atul Paldhikar, Navin Saxena, Ashish Parasrampuria and Sudheer Deshpande who have assisted me in all situations and made my stay at IIT Kanpur a memorable one.

I also thank, collectively and anonymously, all those who have directly or indirectly helped me in the project, but whom I have not mentioned earlier.

This report was produced using LaTeX and texview on the SUN-3 workstations. The source code for the meta-interpreter has been tested in IF/Prolog on the HP-9000 machine and the CLP system has been tested in Quintus Prolog on the SUN-3 workstations. Earlier versions have been tested in Turbo Prolog and WISDOM on PCAT-386 under MS-DOS.

Alexei A. Karve

Contents

List of Figures

Abstract

Available constraint programming languages like Prolog-III, CHIP, CLP(R) and CAL do not completely separate the inference engine and the constraint solver. Substantial work is needed to construct a new CLP system from these existing ones because these languages were constructed for specific problem domains.

A number of general purpose solvers like Mathematica and REDUCE are already available on different kinds of machines. These pre-existing solvers are very good at solving constraints over specific domains for which they are designed. In this report we describe a CLP system which directly uses already available solvers by explicitly separating logical inference from constraint satisfaction.

The constraint solvers are attached separately in the form of independent modules to check the satisfiability of the constraints and find the corresponding solutions. A general interface is defined between the inference engine and the constraint solvers to support pre-existing solvers as well as custom built solvers. A number of independent solvers can be distributed across heterogenous CPUs. A new CLP system can be constructed simply by attaching the required constraint solvers for particular problem domains of interest over the defined interface.

The user of the CLP system specifies the domain and the constraints in his application. The use of constraint solvers is transparent to the user. The constraint solver server on the client side allocates solvers from amongst the available ones to the application as needed. The execution site for the constraint solvers is selected dynamically by the user thus providing both good performance and generality.

Chapter 1

Introduction

In this chapter, we discuss the motivation behind the present work. We also provide a brief overview of work done and finally discuss the organization of the thesis.

1.1 Motivation for Present Work

Constraint logic programming languages like Prolog - III [9, 10], CHIP [12], CLP(R) [15, 16] etc. are tightly coupled with the constraint solvers. These implementations have a fixed structure. Since they were designed with a specific structure in mind, the inference engine and the constraint solvers have not been completely separated. Addition of new domains to such systems is very difficult and time consuming.

Meta-interpreters written in logic programming languages may be used in implementing experimental CLP systems. The meta-interpreter approach [18] simplifies the handling of resolution steps in the CLP language but requires solvers to be written in a logic programming language. Often the solvers are not efficient - and this together with the meta-interpretation overhead produces very slow systems in general.

A Constraint Logic Programming Shell for rapid implementation of CLP systems has been described in [27]. They have discussed the use of special predicates for use by solvers assisting in the construction of a CLP system.

Our aim has been to design an interface for coupling different kinds of solvers to the Inference Engine. The constraints are considered to be built in predicates handled by the constraint solver. Our system is designed to provide the advantages of the meta interpreter approach, without some of the speed drawbacks. We have explicitly separated the logical inference steps and structure satisfiability questions. Our system is different from those mentioned above in that the solvers are attached at run-time rather than having a system with fixed constraint solvers. Since there are many pre-existing solvers already available, we allow these to be incorporated in the CLP system rather than requiring a specially written solver.

1.2 Overview of Work Done

The stages involved in designing the Modular Distributed Constraint Logic Programming System are illustrated in Figure 1.1.

First, a Constraint Logic Programming language was defined and an interpreter was developed on top of IF/Prolog using the meta interpreter approach for domain 'real'. This was used in studying the way new domains could be added to create a general system. In the process, black box constraint solvers were designed and implemented for the domains 'real' and 'boolean'.

A simulator was written in Turbo Prolog for the Vienna Abstract Machine (VAM). Most of the Prolog predicates including the asserta, assertz, retract, univ operator, call and execute were implemented. A list of the internally defined predicates implemented is given in Appendix A. Some examples using these predicates are listed in Appendix C. The VAM simulator was tested by checking the execution of the constraint solvers implemented for IF/Prolog for a number of examples directly on this VAM simulator. An operator precedence parser and code generator were implemented in Turbo Prolog for converting programs to VAM code.

We ported this implementation on to Quintus Prolog on SUN-3 machines from Turbo Prolog because of memory limitations of DOS. After some minor modifications (because of Occur Check), the Constraint Logic Programming Interpreter written earlier, was executed directly on top of the VAM simulator. In this implementation, the solvers were also executing on top of the VAM simulator making the execution very slow. Refer to Figure 1.2. The exact structure was as follows : first the VAM-simulator in Quintus Prolog, second the meta interpreter executing on the VAM-Simulator along with the constraints solvers and third the programs meant to be executed on the meta interpreter (e.g. The Pigeons and Rabbits Problem, Installments Problem, Light Meals Problem, etc).

We modified and extended the VAM to define a structure independent inference engine as shown in Figure 1.3 which currently forms the core of our CLP system. A separate compiler was written in C (YACC and LEX) for compiling the Prolog Programs with constraints to Vienna Abstract Machine Code. In the current implementation, the code produced by this compiler for clauses written by a user is loaded by the simulator. The Prolog compiler is however used to compile user queries when the user interacts with the system. The constraint solvers mentioned in the above paragraph were separated from the Constraint Logic Programming Meta Interpreter and were directly executed on top of Quintus Prolog as shown in Figure 1.3. These are now accessed through an interface thus increasing the execution speed.

The first version of the implementation of the CLP system is operational. To check the working of the interface for different kinds of solvers, we have also implemented an incremental solver for domain 'real'. The novelty of our CLP system in its current form is that new solvers can be introduced in a completely modular and incremental way. They could even be running on a machine other than the one on which the inference engine is running. A Constraint Solver Server and an Advice

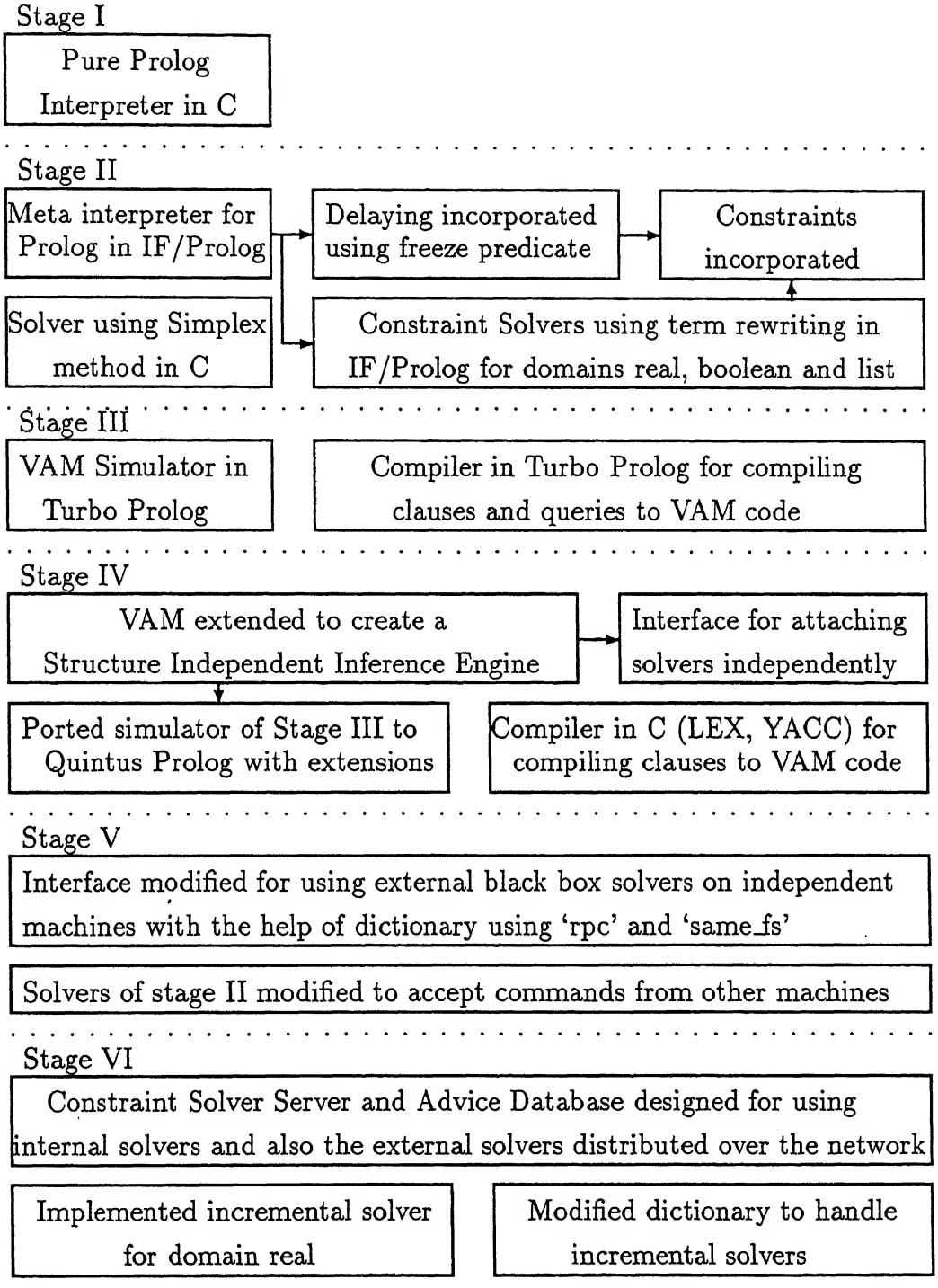


Figure 1.1: Stages in the design of the modular distributed CLP system

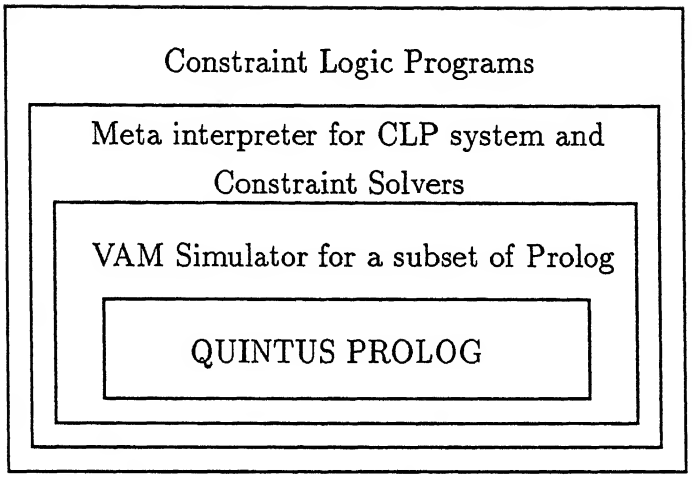


Figure 1.2: Meta Interpreter for CLP system

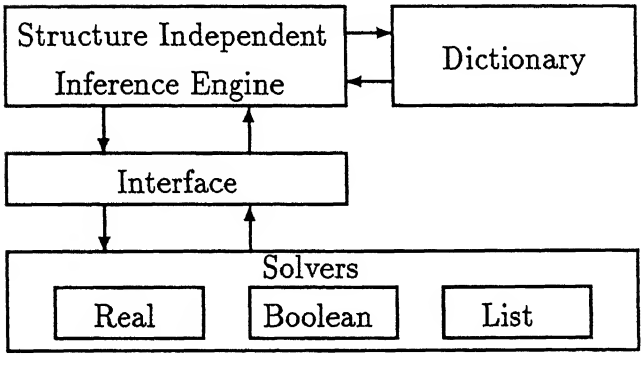


Figure 1.3: Using Internal Solvers

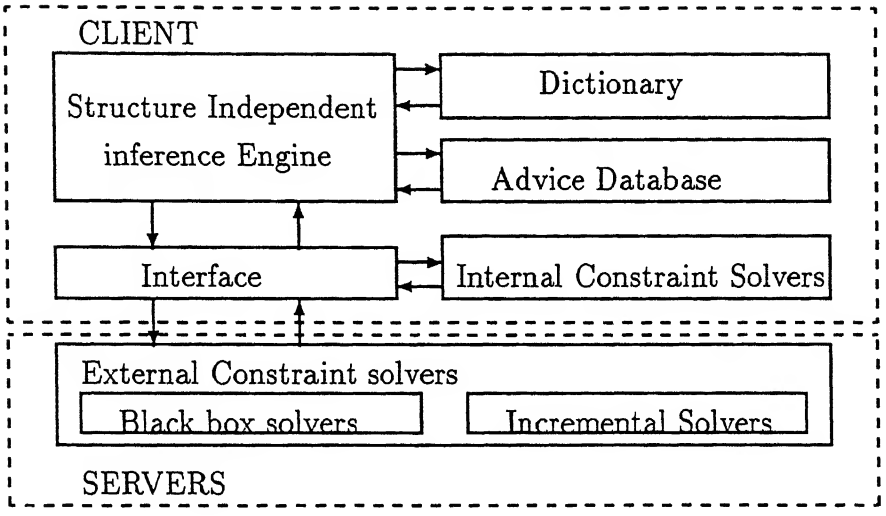


Figure 1.4: Using External Solvers

Database have been designed. The routing of the constraints to their respective solvers is handled by the *constraint solver server* which is transparent to the user by the use of an *advice database*, where information about solvers is maintained. This is illustrated in Figure 1.4.

1.3 Outline of Thesis

The rest of the thesis is organised as follows :

A brief review of Prolog, Logic Programming and Constraint Logic Programming is presented in Chapter 2. The meta-interpreter approach used for designing a CLP system is described in Chapter 3. The details of the CLP system meta-interpreter implemented earlier as a step towards building the final system have been explained in this chapter. The facilities provided in the final system to write constraint logic programs have been explained with examples in Chapter 4. In Chapter 5, we have given an overview of the Vienna Abstract Machine. We have also covered the implementation aspects of the Vienna Abstract Machine on top of Prolog. We have described the Inference Engine in detail in Chapter 6 where extensions made to the VAM are also discussed. The compilation aspects have been covered in Chapter 7. The different kinds of Solvers have been dealt with in Chapter 8. The Solvers to be interfaced to the Inference Engine may be present either on the client machine or any of the server machines. This chapter deals with the invocation details and backtracking in the solvers. In Chapter 9, the Constraint Solver Server present on the client machine has been detailed. Its work is to make use of the advice database for allocation of constraint solvers. The Interface has also been explained in this chapter. Adding of solvers is illustrated with the help of examples in Chapter 10. Finally we have listed the applications and made suggestions for further work in Chapter 11.

Chapter 2

Review of Constraint Logic Programming

2.1 Logic Programming and Prolog

Prolog, an acronym for Programming in Logic, was created at the Faculty of Sciences at Luminy in Marseilles, France. The creators' original objective was to integrate a development in mathematical logic – the resolution principle proposed by Robinson into a programming language. Robinson's resolution principle suggested the application of only one powerful rule of inference to mechanical theorem proving. This facilitated the design of a programming language that would enable a programmer to make a computer simulate the thinking process by making deductions from information given in logical formulas. A tutorial of a succession of proposed logic programming schemes has been discussed in [6]. Prolog has been considered a model for designing the computers of the future [8].

Logic Programming is very appropriate to state constraint search problems. Its relational form and the logical variables are very appropriate to formulate such problems in a declarative way and its nondeterministic computation liberates the user from tree search programming. However, Prolog is too inefficient to tackle large search problems.

Prolog has at least two attractive features to be used in constraint problems : expressiveness and unification. However, its simple backtrack search and its lack of data-driven computation makes Prolog inefficient in most cases. The general computational paradigm of Prolog is "generate-and-test". This is elaborated in Chapter 4.

2.2 Constraints

Given a computation domain, a constraint expresses a relationship between some objects of this domain. Constraint manipulation and propogation have been studied in the Artificial Intelligence community in the late 70's and early 80's especially in

the USA. They provide very interesting problem solving techniques like local value propogation, data driven computation, sophisticated search algorithms (e.g. forward checking) and consistency checking. The general idea behind these techniques is the use of constraints to prune the search space in an 'a priori' way, i.e. before the generation of values.

The most outstanding feature of constraint programming [26] is that it allows the declarative description of problems which are solved by indicating a goal without reference to the method by which it should be established. Constraints in a program are evaluated automatically and affect the execution of the program depending on their meaning.

Three operations can be distinguished on constraints :

1. *Constraint Formulation* is the adding of new constraints as commitments in the design process. A problem solver that can introduce new constraints need not work with all of the details at once. This idea is consistent with the common experience of working on problems that are imprecisely formulated, but which become more tightly specified during the solution process. In contrast, the traditional constraint satisfaction approach works with a fixed number of constraints that are all known in the beginning.

2. *Constraint Propogation* is the creation of new constraints from old constraints. When constraints are propogated, they bring together the requirements from separate parts of the problem. Constraint propogation makes possible a *least-commitment* strategy of deferring decisions for as long as possible. The problem solver works to keep its options open, and reasons by elimination when constraints from other subproblems become known.

3. *Constraint Satisfaction* is the operation of finding values for variables so that a set of constraints on the variables is satisfied. When the constraints and variables come from different subproblems, constraint satisfaction plays a coordinating role by pooling the constraints and intersecting their solutions.

2.3 Constraints in Logic Programming

Constraints are incorporated into logic programming [11, 7, 19, 24] to increase the descriptive power of logic programming. Extensive work has been conducted in this direction especially in Europe (Marseille, European Computer- Industry Research Center), USA (IBM Yorktown Heights), Canada (Vancouver) and Australia (Monash University). The general idea behind the introduction of these extensions inside logic programming is the use of some mathematical tools to solve numerical constraints and the use of consistency checking and constraint propogation techniques to solve symbolic constraints. Active use of constraints helps in reducing the search space of large search problems as much as possible. The advantage of constrained terms is the increase in efficiency obtained by testing the constraints as soon as possible.

Prolog-II [13] provides delaying of subgoals with the freeze/2 predicate. Thus, freeze(X,P) is the same as call(P), except that P will be executed only when X is bound to a non-variable term. X is called the freezing variable and P the frozen goal. Lot of work has been done in order to provide a *coroutining* mechanism which allows modification of the computation (control) rule. Refer to Appendix C.9 for use of freeze predicate in this manner.

Prolog-III [10, 9] uses a simplex-like algorithm to solve linear equations and inequations on rational numbers for linear programming purposes. It also provides a saturation method to deal with boolean terms.

CLP(R) [15] handles linear equations and inequations over real numbers. A modification of the standard simplex algorithm which serves as a CLP(R) constraint solver has been presented in [21].

CHIP [12] works on finite domain restrictive terms, boolean terms and linear rational terms.

CAL (Constrainte Avec Logique) [2] supports the writing and solving of linear and nonlinear algebraic polynomial equations, boolean equations and linear inequalities as constraints.

2.4 The CLP scheme

The CLP scheme [20] is not an extension to logic programming but provides a general framework from which Prolog extensions can be derived. The unification algorithm, the heart of a Prolog interpreter, is itself a particular case of constraint solving. It tells us whether two terms, such as f(x,a) and f(b,y), can be made identical by a particular instantiation of the variables (here x=b and y=a). In other words it decides whether the equation f(x,a)=f(b,y) is solvable and in the process outputs a substitution that explicitly represents the set of all solutions. This finite explicit representation although a strength of unification, is also its weakness since for many domains of interest, such finite explicit representations do not exist, and therefore the concept of unification is not sufficient.
Consider a simple Prolog program to compute factorial as shown below :

fact(N,F) :- $N > 0$,
fact($(N-1)$,M),
$F = N * M$.
fact(0,1).

A query such as :

```
?- fact(3,F).
```

returns the answer $F = 6$.
To achieve this answer, Prolog goes through a number of steps, which can easily be expressed in terms of solving constraints.
Four sets of constraints are obtained :

$C1 = \{3 > 0, F = 3 * M1\}$
$C2 = C1$ and $\{2 > 0, M1 = 2 * M2\}$
$C3 = C2$ and $\{1 > 0, M2 = 1 * M3\}$
$C4 = C3$ and $\{0 = 0, M3 = 1\}$

The first three sets of constraints come from the first clause of the program and have no explicit solutions. The fourth set however can be solved using the second clause, and its solution provides the answer.

Thus a Prolog program can be thought of as a constraint-solving problem. A CLP program for factorial will be same as above. However, there is a fundamental difference between a normal Prolog program and a CLP-program. In the case of the Prolog program, if the factorial function is reversed, i.e., asking for the number whose factorial is say 6,

```
?- fact(N,6).
```

the subtraction operation in Prolog works only if its arguments are instantiated, and in this case the factorial program contains the expression $N - 1$, where N has no value. In CLP, however, this problem does not arise and we get the expected answer. Note that it is possible to rewrite the Prolog program without this operator by using the successor function ($s(X) = X + 1$), this creates different problems of checking for all possible values. Refer to Appendix F.1 for the factorial program using constraints along with all test cases for using the same.

Solving a goal is defined in terms of solvability of the set of constraints and not in terms of finding a value. The execution steps of CLP programs depend upon whether or not a constraint is satisfiable in a given domain.

Chapter 3

Meta-Interpreter for Constraint Logic Programming

Meta-programs treat other programs as data. They analyze, transform, and simulate other programs. The writing of meta-programs, or meta-programming is particularly easy in Prolog due to the equivalance of programs and data : both being Prolog Terms.

A Meta-interpreter for a language is an interpreter for the language written in the language itself. The ability to write a meta-interpreter in Prolog enables the building of an integrated programming environment for CLP by giving access to the computational process of Prolog.

We describe in this chapter, the meta interpreter we have developed in IF-Prolog before discussing the CLP system implementation on an Abstract Machine. In the meta-interpreter, we directly use all the system predicates provided by IF-Prolog. We have provided predicates for delaying of goals and for specifying constraints. The clauses are specified by the user directly in IF/Prolog.

3.1 Meta interpreter for Prolog

The Prolog procedure *solve* uses the built in predicate *clause(Goal,Tail)*, which determines the (first) head of a Prolog rule that unifies with *Goal* and binds *Tail* to the tail of that rule. In the case of unit clauses, *Tail* is bound to *true*. The meta-interpreter takes the form :

```
solve(true).
solve([Goal|RestGoal]) :- solve(Goal), solve(RestGoal).
solve(Goal) :- clause(Goal,Tail), solve(Tail).
```

3.2 Incorporating Freeze

To incorporate *freeze*, two additional parameters are needed: the list representing a current *Freezer* and another list representing its modified counterpart, the

NewFreezer. Both lists contain pairs *(variable,goal)* in which *variable* is an unbound (or frozen) variable and the *goal* is a term to be activated as a procedure as soon as the variable becomes bound (or unfrozen). Immediately after a clause matches a Goal in the database, the defrost predicate is used to check whether any variable has thawed, in which case the corresponding goal is immediately solved after updating the Freezer. This is shown below :

```
solve(true,Freezer,Freezer).
solve([Goal|RestGoal],Freezer,NewFreezer) :-
   solve(Goal,Freezer,TempFreezer),
   solve(RestGoal,TempFreezer,NewFreezer).
solve(Goal,Freezer,NewFreezer) :-
   clause(Goal,Tail),defrost(Freezer,TempFreezer),
   solve(Tail,TempFreezer,NewFreezer).
solve(freeze(X,Goal),Freezer,[[X|Goal]|Freezer]) :-
   var(X).
solve(freeze(X,Goal),Freezer,NewFreezer) :-
   nonvar(X),solve(Goal,Freezer,NewFreezer).
defrost([],[]).
defrost([[X|Goal]|Freezer],[[X|Goal]|NewFreezer]) :-
   var(X),defrost(Freezer,NewFreezer).
defrost([[X|Goal]|Freezer],NewFreezer) :-
   nonvar(X),defrost(Freezer,TempFreezer),
   solve(Goal,TempFreezer,NewFreezer).
```

Notice that in the backtracking mode, thawed procedures should be refrozen.

3.3 Disequalities

Freeze can be used in "simulating" another useful Prolog extension : *dif(X, Y)* or *X* \== *Y*. Although this predicate is available on most interpreters, it is applicable only when both X and Y are bound. If even one of the variables is unbound, the interpreters return failure. The more general *dif* could be programmed using the clause :

```
dif(X,Y) :- freeze(X, freeze(Y, different(X,Y))).
```

in which *different(X, Y)* would test whether or not the bound variable X is different from the bound variable Y.

3.4 Incorporating constraints

Two more parameters : the list representing the current set of constraints *PreviousConstraints* and another representing the new list of constraints *NewConstraints* are incorporated. The listing of the meta interpreter for our CLP system is shown below :

```
/* metaclp.pro */
/* CLP meta-interpreter in IF/Prolog on HP-9000 machine */

:- [constraint]. % calling the constraint solvers
:- [xprn].       % float constraint solver
:- [floateqn].
:- [booleaneqn]. % boolean constrant solver
:- [listeqn]     % list constraint solver

solve(true,FrozenGoals,FrozenGoals,Constr,Constr) :-
   !. /* for unit clause */
solve(','(Goal1,Goal), FrozenGoals, NewFrozenGoals,
      PreviousConstraints,NewConstraints) :- !,
   solve(Goal1,FrozenGoals,TmpFrozenGoals,
         PreviousConstraints,Tmp1Constraints),
   solve(Goal,TmpFrozenGoals,NewFrozenGoals,
         Tmp1Constraints,NewConstraints).
solve(freeze(X,Goal),FrozenGoals,[[X|Goal]|FrozenGoals],
      Constraints,Constraints) :- var(X),!.
solve(freeze(X,Goal),FrozenGoals,NewFrozenGoals,
      PreviousConstraints,NewConstraints) :-
   nonvar(X),!,
   solve(Goal,FrozenGoals,NewFrozenGoals,
         PreviousConstraints,NewConstraints).
solve(Goal,FrozenGoals,NewFrozenGoals,
      PreviousConstraints,NewConstraints) :-
   clause(Goal,Tail),
   solve_thawed(FrozenGoals,TmpFrozenGoals,
                PreviousConstraints,TmpConstraints),
   convert(Tail,TmpFrozenGoals,NewFrozenGoals,
           TmpConstraints,NewConstraints,ReplacedTail),
   ReplacedTail.
solve(Goal,FrozenGoals,FrozenGoals,Constraints,Constraints) :-
/* This rule is required for preventing the flow of control to
   go to the next "solve" if  all user defined clauses of Goal
   (if present) have failed.  If no user defined  clauses  are
   present,  the  control passes  over to  the next "solve" to
   execute the goal  as a normal prolog Goal.  The cut in this
   "solve" after "clause" cannot  be  present in above "solve"
   after  "clause"  since  it  prevents normal backtracking of
   "clause".
*/
   clause(Goal,Tail),!,fail.
solve(Goal,FrozenGoals,FrozenGoals,
```

```
      Constraints,Constraints) :- call(Goal).

/* Solve the goals which have got unfrozen */
solve_thawed([],[],Constraints,Constraints):-!.
solve_thawed([[X|Goal]|FrozenGoals],
             [[X|Goal]|NewFrozenGoals],
             PreviousConstraints,NewConstraints) :-
   var(X),!,
   solve_thawed(FrozenGoals,NewFrozenGoals,
                PreviousConstraints,NewConstraints).
solve_thawed([[X|Goal]|FrozenGoals],NewFrozenGoals,
             PreviousConstraints,NewConstraints) :-
   nonvar(X),!,
   solve_thawed(FrozenGoals,TmpFrozenGoals,
                PreviousConstraints,TmpConstraints),
   solve(Goal,TmpFrozenGoals,NewFrozenGoals,
         TmpConstraints,NewConstraints).

convert((!,X),OldFrozenGoals,NewFrozenGoals,
        OldConstraints,NewConstraints,
        ','(call(!),NewX)) :- !,
   convert(X,OldFrozenGoals,NewFrozenGoals,
           OldConstraints,NewConstraints,NewX).
convert((merge_constraints(C),X),OldFrozenGoals,NewFrozenGoals,
        OldConstraints,NewConstraints,
        (append(OldConstraints,C,Tmp1Constraints),
         solve_constraint(Tmp1Constraints,Tmp2Constraints),
         NewX
       )) :- !,
   convert(X,OldFrozenGoals,NewFrozenGoals,
           Tmp2Constraints,NewConstraints,NewX).
convert((write_constraints,X),OldFrozenGoals,NewFrozenGoals,
        OldConstraints,NewConstraints,
        (write('Debugging ...'),nl,writeList(OldConstraints),
         NewX
       )) :- !,
   convert(X,OldFrozenGoals,NewFrozenGoals,
           OldConstraints,NewConstraints,NewX).
convert((X1,X),OldFrozenGoals,NewFrozenGoals,
        OldConstraints,NewConstraints,
        (solve(X1,OldFrozenGoals,FrozenGoals,
               OldConstraints,Constraints), NewX)) :-
   nonvar(X1),nonvar(X),!,
   convert(X,FrozenGoals,NewFrozenGoals,Constraints,
```

```
            NewConstraints,NewX).
convert(!,FrozenGoals,FrozenGoals,
   Constraints,Constraints,!) :- !.
convert(true,FrozenGoals,FrozenGoals,
        Constraints,NewConstraints,
        solve_constraint(Constraints,NewConstraints)
        ) :- !.
convert(write_constraints,FrozenGoals,FrozenGoals,
        Constraints,Constraints,
        (write('Debugging ...'),nl,writeList(Constraints))
       ) :- !.
convert(merge_constraints(C),FrozenGoals,FrozenGoals,
        Constraints,NewConstraints,
        (append(Constraints,C,TmpConstraints),
         solve_constraint(TmpConstraints,NewConstraints)
       )) :- !.
convert(X,OldFrozenGoals,NewFrozenGoals,
        OldConstraints,NewConstraints,
        solve(X,OldFrozenGoals,NewFrozenGoals,
              OldConstraints,NewConstraints)
   ) :- nonvar(X),!.

solve_goal(Goal) :-
   convert(Goal,[],NewFrozenGoals,[],NewConstraints,NewGoal),
   NewGoal,
   write('Unsolved Frozen Goals :\n'),
   write(NewFrozenGoals),nl,
   write('Unsolved Constraints:\n'),
   writeList(NewConstraints).
```

The main call to the interpreter is through solve_goal. It starts by converting the user specified *Goal* into a proper format which IF/Prolog can understand (converting the call for merge_constraints to calling of the constraint solver) and executes it.

The relation *solve(Goal, ...)* is true if *Goal* is true with respect to the program being interpreted. The interpreter can be read as follows. For the first clause of solve, the constant *true* is true. The solve fact states that the empty goal, represented in Prolog by atom *true*, is solved. The conjunction *','(Goal1,Goal)* is true if *Goal1* is true and *Goal* is true. The next two clauses are used to interprete the freeze predicate. The predicate freeze is used to delay *Goal* on variable *X* in *freeze(X,Goal)*. If *X* is not instantiated, then *Goal* is frozen on the FrozenGoals Stack to be thawed later when X gets instantiated. If however, X is already instantiated, then *Goal* is solved directly. The next clause for solve shows that *Goal* is true if there is a clause Goal :- Tail such that Tail is true. This means, choose a clause from the program whose head unifies with the goal, and recursively solve the body of the clause. The final clause for solve states that if Goal is a predicate internally

defined by the interpreter on which the meta-interpreter is running, then allow the interpreter to execute it.

This demonstrates that the meta-interpreter indeed reflects Prolog's choices of implementing the abstract computation model of logic programming. The two choices are the selection of the leftmost goal to reduce, and sequential search and backtracking for the nondeterministic choice of the clause to use to reduce the goal. The goal order of the body of the *solve* clause handling conjunctions guarantees that the leftmost goal in the conjunction is solved first. Sequential search and backtracking comes from Prolog's behaviour in satisfying the *clause* goal.

The call to *clause* in the fifth clause for predicate solve performs the unification with the heads of the clauses appearing in the program. It is also responsible for giving different solutions on backtracking. Backtracking also occurs in the conjunctive rule reverting from Goal to Goal1.

The *solve_thawed* predicate is for solving the frozen goals. After every check of a clause, the frozen predicates are checked for possible execution. If the variable on which a goal is frozen has now been instantiated, then the goal is set up for execution. If the variable is not instantiated, then the goal is pushed back on to the FrozenGoals Stack.

Simulating the behaviour of cut is a problem with a meta interpreter. The naive solution is to consider cut as a system predicate. Effectively, this means adding the clause :

```
solve(!) :- !.
```

This clause however does not have the required effect. The cut in the clause guarantees commitment to the current solve clause rather than affecting the search tree of which the cut is a part. In other words, the scope of the cut is too local. The solution we have provided is to convert the right hand sides of clauses used, to calls, and executing them such that the cut takes place at the ancestor.

The first clause of convert with *(!,X)* keeps the cut transparent by converting it to a *call(!)*. The second clause of convert with *(merge_constraints(C),X)* is used to convert this merge_constraints call to calls for appending the current set of constraints with the old set of constraints using *append* and for solving them with *solve_constraint*. The third clause of convert with *(write_constraints,X)* is converted to calls to inform the user of the current set of constraints. The fourth clause is the conjunction *(X1,X)* which is converted to calls to the *solve* predicate. The goal *!* is transparently converted to a *!*. The goal *true* is converted to a call to *solve_constraint* since in the current implementation the constraints are solved at the ends of derivations.

3.5 Invoking the constraint solvers

The proper constraint solvers are called depending on the name of the solver specified by the user. The default constraint solver used is for domain float, namely solve_float_equation. A part of the listing is shown below.

```
/* constraint.pro */
/* Procedure to solve the list of constraints */
solve_constraint(A,C) :-
   actual_solve_constraint(A,B),
   continue_solve_constraint(B,C).
continue_solve_constraint(A,B) :-
   retract(constraint_changed),!,
   solve_constraint(A,B).
continue_solve_constraint(A,A).

assert_if_not_present(A) :-
   retract(A),fail.
assert_if_not_present(A) :-
   asserta(A).

actual_solve_constraint([],[]).
actual_solve_constraint(
      [([Type],Solver,SIn1)|SIn],NewSOut) :- !,
   S =.. [Solver, SIn1, SOut1],
   S,
   actual_solve_constraint(SIn,SOut),
   cons2_constraint(Type, Solver, SOut1, SOut, NewSOut).
% default float constraint solver
actual_solve_constraint([SIn1|SIn],NewSOut) :-
   actual_solve_constraint(
      [([float],solve_float_equation,SIn1)|SIn],NewSOut).

cons2_constraint(_, _, [], SOut, SOut) :- !.
cons2_constraint(ConstraintType, ConstraintSolver,
      SOut1, SOut,
      [([ConstraintType],ConstraintSolver,SOut1)|SOut]).
```

actual_solve_constraint calls the solver with a single constraint at a time, one by one until all the constraints have been considered. If during *actual_solve_constraint*, any of the constraints are modified, as indicated by the *constraints_changed* flag, then *continue_solve_constraint* again marks the complete set of constraints as 'to be solved'. *solve_constraint* keeps calling *actual_solve_constraint*, until none of the constraints are modified.

3.6 Using the meta-interpreter for CLP

Consider the program in Figure 3.1 written for the meta-interpreter. The first two clauses represent Ohm's law and Kirchoff's law; the next two define the sum of a list in the usual way using head and tail notation. This is followed by a small database

```
:- [metaclp].       % Load the CLP meta-interpreter
ohmlaw(V,I,R) :-
   merge_constraints([V=I*R]).
kirchoff(L) :-
   sum(L,Zero),merge_constraints([Zero=0.0]).
sum([],Zero) :- merge_constraints([Zero=0.0]).
sum([H|T],N) :-
   merge_constraints([H+M=N]),sum(T,M).
availres(10.0).
availres(14.0).
availres(27.0).
availres(60.0).
availres(100.0).
availcell(10.0).
availcell(20.0).
constrainV(V) :-
   merge_constraints([14.5<V, V<16.25]).
goal :- solve_goal((
      constrainV(V2),
      availres(R1), availres(R2), availcell(V),
      ohmlaw(V1,I1,R1), ohmlaw(V2,I2,R2),
      kirchoff([I1,-(I2)]),
      kirchoff([-(V), V1, V2]),
      write('V='),write(V),
      write(', R1='),write(R1),
      write(', R2='),write(R2),
      write(', V2='),write(V2),
      sum_of_products(V2,NewV2),
      write(' = '),write(NewV2),nl,fail
   )).
```

Figure 3.1: Problem using Ohm's Law and Kirchoff's Law

of resistors and cells. Finally we have the goal which calls *solve_goal* to start the meta-interpreter. The parameter to *solve_goal* consists of the representation of a simple circuit of two resistors (R1 and R2) connected in series with a cell (V). It is required that the voltage over R2 should be between 14.5 and 16.5 volts. Thus the goal asks for the possible values of the components in this list.

Each instance of a,b,c of R1, R2, and V such that $14.5 < V2 < 16.25$ gives rise to the problem : Is the following solvable?

$V1/a - V2/b = 0.0$

$V1 + V2 = c$

The meta interpreter computes the three sets of solutions:

```
V=20.0, R1=10.0, R2=27.0, V2=27.0 * 0.540541 = 14.5946
V=20.0, R1=14.0, R2=60.0, V2=60.0 * 0.27027 = 16.2162
V=20.0, R1=27.0, R2=100.0, V2=100.0 * 0.15748 = 15.748
```

Note how the constraints are specified in the form of a list passed as a parameter to the predicate *merge_constraints* in Figure 3.1. Also note that a special call to the predicate *sum_of_products* needs to be given to simplify the expression given by the solver. Further details regarding the requirement of this special predicate are given on page 54. In the definition of *sum*, the variable M is used by the constraint $H + M = N$ which is defined only by the next term, *sum(T,M)*. This would not be possible with most Prolog implementations if the constraint is specified directly because variables have to be instantiated before any expression containing them is evaluated.

Chapter 4

Solving Problems on our system

In this chapter we present formulas which describe our system and also give an idea of how the predicate *freeze* and the *constraints solving techniques* provided can be used for solving generate and test type of problems, and for geometrical theorem proving. For further examples demonstrating both the capability to delay constraints until they are consistent and the equation solving ability of the constraint solvers to find solutions, see Appendix F.

4.1 Meaning of a program in our system

A deducible fact represents a proposition that the programmer considers true. The set of deducible facts of a program is usually infinite. The execution of a program can be regarded as a search through this infinite database. This database cannot be stored in explicit form, but must be finitely represented from which all information in the database can be deduced. The aim of a program's execution is to solve the following problem : Given a sequence of terms and a system of constraints, find the values of the variables that satisfy all the constraints and transform the sequence of terms into a sequence of deductions.
The three formulas below describe our CLP system :

1. (W, t_0 t_1 ... t_n, S),
2. s_0 -> s_1 ... s_m, R
3. (W, s_1 ... s_m t_1 ... t_n, S U R U $\{s_0 = t_0\}$).

Formula 1 represents the state of the machine at any given moment, W the set of variables whose values we want to establish, t_0 ... t_n is a sequence we are trying to delete and S is a system of constraints that has to be satisfied. Formula 2 changes the state of the machine. If necessary, some variables have to be renamed. Formula 3 is the new state of the machine after rule 2 has been applied. We start from an

```
installments_capital([],0.0).
installments_capital([I|X],C) :-
   {Y=1.1*C-I}, installments_capital(X,Y).
```

Figure 4.1: Banking Calculation Problem

initial state (where W is the set of variables that appear in the query) and then calculate all the states by applying the above procedure. Each time we arrive at a state where the sequence of terms is empty, we simplify the system of constraints with which it is associated and provide this as an answer.

This is illustrated using the Banking Calculation Problem [24] listed in Figure 4.1. In this example, the task is to calculate a series of successive installments that have to be made to repay an amount of money borrowed from a bank. Assume a constant time period between installments and a 10 percent interest rate per period.

The first rule expresses the fact that it is not necessary to pay an installment if the amount owed is 0.0. The second recursive rule states that a list of installments required to repay an amount C consists of an installment I plus the list of installments required to repay the amount C increased by 10 percent interest and reduced by installment I.

This program can be used in different ways. The most spectacular way is to ask what three payments should be made to repay Rs 1000.0 such that the second payment is twice the value of the first, and the third payment is three times the value of the first. In other words, what value of I is required to have the sequence of installments [I,2I,3I]. The following query is all that needs to be specified by the user :

```
goal :- installments_capital([I,2.0*I,3.0*I],1000.0),
        write("Answer:   I="),write(I),nl.
```

A trace of the program is as follows :

$(\{I\}$, installments_capital($[I,2*I,3*I]$,1000.0),$\{\})$.

By applying the second rule of the program, we get :

$(\{I\}$, installments_capital(X,$1.1*C-NewI$),

$\{$installments_capital($[I,2*I,3*I]$,1000.0) =

installments_capital($[NewI—X]$,C)$\}$).

We obtain this by giving the variables in the rule all the possible values that satisfy the system. The particularized rules do not use variables or constraints and form the deductible facts of the program. This simplifies to :

$(\{I\}$, installments_capital(X,$1.1*C-NewI$),

$\{NewI = I,\ X = [2*I,3*I],\ C$=1000.0$\}$).

and then to :

$(\{I\}$,installments_capital($[2*I,3*I]$,$1100.0-I$),$\{\})$.

```
mortgage(P,Time,I,B,MP) :-
   {Time=<1.0, B+MP = P*(1.0+I)}.
mortgage(P,Time,I,B,MP) :-
   {1.0<Time},
   mortgage(P*(1.0+I)-MP, Time-1.0, I, B, MP).
goal :-
   Time = 5.0, I=0.1, B=0.0, mortgage(P,Time,I,B,MP),
   write('P='),write(P),nl,write('MP='),write(MP),nl.
```

Figure 4.2: Mortgage

By applying the second rule and third rule again, we eventually obtain,
$(\{I\},\{\}, \{1331.0 - 6.41 * I = 0.0\})$.
Thus the sequence of terms has been transformed successively into a sequence of deductible facts by satisfying all the constraints. This leads to the final response of :
$\{I = 207.64\}$.
Thus I = Rs 207.64.

Our CLP system also has the potential to produce symbolic output as illustrated by the next example listed in Figure 4.2. The predicate *mortgage* is defined in terms of the principal (P), the duration (Time), the intertest rate (I), the monthly payments (MP), and the balance (B). The goal specifies values for the variables Time, I, and B, leaving the CLP system to determine the other two.

The answer, $P = 3.79079 * MP$ for *goal* expresses the linear relation that exists between principal amount and monthly payment.

The practical realization of the three formulas enumerated in this section is considered in the following chapters.

4.2 The generate-and-test technique

Generate-and-test is a common technique in algorithm design and programming. It is easy to write logic programs that under the execution model of Prolog, implement the generate-and-test technique. In generate-and-test, one process or routine generates the set of candidate solutions to the problem, and another process or routine tests the candidates trying to find one, or all of the candidates which actually solve the problem. Typically, generate-and-test programs are easier to construct than programs that compute the solution directly, but they are also less efficient. A standard technique for optimizing generate and test programs is to try and "push" the tester inside the generator, as "deep" as possible. Ultimately, the tester is completely intertwined with the generator, and only correct solutions are generated to begin with.

Generate-and-test programs in Prolog typically have a conjunction of two goals,

```
mortgage(P,Time,I,B,MP) :-
   {Time=<1.0, B+MP = P*(1.0+I)}.
mortgage(P,Time,I,B,MP) :-
   {1.0<Time},
   mortgage(P*(1.0+I)-MP, Time-1.0, I, B, MP).
goal :-
   Time = 5.0, I=0.1, B=0.0, mortgage(P,Time,I,B,MP),
   write('P='),write(P),nl,write('MP='),write(MP),nl.
```

Figure 4.2: Mortgage

By applying the second rule and third rule again, we eventually obtain,
$(\{I\},\{\}, \{1331.0 - 6.41 * I = 0.0\})$.
Thus the sequence of terms has been transformed successively into a sequence of deductible facts by satisfying all the constraints. This leads to the final response of :
$\{I = 207.64\}$.
Thus I = Rs 207.64.

Our CLP system also has the potential to produce symbolic output as illustrated by the next example listed in Figure 4.2. The predicate *mortgage* is defined in terms of the principal (P), the duration (Time), the intertest rate (I), the monthly payments (MP), and the balance (B). The goal specifies values for the variables Time, I, and B, leaving the CLP system to determine the other two.

The answer, $P = 3.79079 * MP$ for *goal* expresses the linear relation that exists between principal amount and monthly payment.

The practical realization of the three formulas enumerated in this section is considered in the following chapters.

4.2 The generate-and-test technique

Generate-and-test is a common technique in algorithm design and programming. It is easy to write logic programs that under the execution model of Prolog, implement the generate-and-test technique. In generate-and-test, one process or routine generates the set of candidate solutions to the problem, and another process or routine tests the candidates trying to find one, or all of the candidates which actually solve the problem. Typically, generate-and-test programs are easier to construct than programs that compute the solution directly, but they are also less efficient. A standard technique for optimizing generate and test programs is to try and "push" the tester inside the generator, as "deep" as possible. Ultimately, the tester is completely intertwined with the generator, and only correct solutions are generated to begin with.

Generate-and-test programs in Prolog typically have a conjunction of two goals,

```
sort(Xs,Ys) :-
   genlengthlist(Xs,Ys),
   ordered(Ys),write('Ordered'),nl,
   permutation(Xs,Ys).
permutation(Xs,[Z|Zs]) :-
   select(Z,Xs,Ys), permutation(Ys,Zs).
permutation([],[]).
select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).
ordered([X]).
ordered([X,Y|Ys]) :-
   freeze(X,freeze(Y,X=<Y)),ordered([Y|Ys]).
genlengthlist([],[]) :- !.
genlengthlist([_|Rest],[_|VarRest]) :-
   genlengthlist(Rest,VarRest).
```

Figure 4.3: Permutation Sort

in which one acts as the generator and the other tests whether the solution is acceptable, as in the following clause :

```
find(X) :- generate(X),test(X).
```

When called with *find(X)*, *generate(X)* succeeds returning some X, with which *test(X)* is called. If the test goal fails, execution backtracks to *generate*, which generates the next element X. This continues iteratively until the tester successfully finds a solution with the distinguishing property, or the generator exhausts alternative solutions. Reference [31] shows a number of examples using this technique.

4.2.1 Using a delay mechanism (freeze or dif)

Consider the example "permutation sort". The top level is as follows :

```
sort(Xs,Ys) :- permutation(Xs,Ys),ordered(Ys).
```

Abstractly, this program nondeterministically guesses the correct permutation via *permutation(Xs, Ys)*, and *ordered(Ys)* checks that *Ys* is actually ordered. Permutation sort is a highly inefficient sorting algorithm, requiring time exponential in the size of the list to be sorted. The generator for permutation sort, *permutation* selects an arbitrary element and recursively permutes the rest of the list. The tester, *ordered*, verifies that the first two elements of the permutation are in order, then recursively checks the rest. If however, the order of *permutation* and *ordered* is interchanged from the one shown above, then only the correctly ordered elements are selected. The freeze predicate as defined by Prolog-II is used to preorder the output list in the program shown in Figure 4.3.

```
queens(N,Qs) :-
   range(1,N,Ns,Qs),safe(Qs),permutation(Ns,Qs).

safe([Q|Qs]) :- safe(Qs), notattack(Q,Qs).
safe([]).

notattack(X,Xs) :- notattack(X,1,Xs).

notattack(_,_,[]) :- !.
notattack(X,N,[Y|Ys]) :-
   freeze(X, freeze(Y, X \== Y + N)),
   freeze(X, freeze(Y, X \== Y - N)),
   N1 is N + 1,
   notattack(X, N1, Ys).

permutation(Xs,[Z|Zs]) :-
   select(Z,Xs,Ys), permutation(Ys,Zs).
permutation([],[]).

select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).

range(M,N,[M|Ns],[_|Temp]) :-
    M < N, M1 is M + 1, range(M1,N,Ns,Temp).
range(N,N,[N],[_]).
```

Figure 4.4: N Queens Problem

The output produced by sort([1, 3, 5, 2, 7, 9, 0, 8, 6, 4],Z) is Z = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].

Let us consider another problem, the N queens problem which requires the placement of N queens on an N-by-N rectangular board so that no two queens are on the same line : horizontal, vertical or diagonal.

The program has been well studied in the recreational mathematics literature. There is no solution for N=2 and N=3. Given in Figure 4.4 is a generate-and-test program using the freeze predicate for generating only the necessary permutations.

The relation *queens(N,Qs)* is true if Qs is a solution to the N Queens Problem. Solutions are specified as a permutation of the list of numbers 1 to N. The first element of the list is the row number to place the queen in the first column, the second element indicates the row number to place the queen in the second column, etc.

The program behaves as follows. The predicate *range* creates a list Ns of the numbers 1 to N and an uninitialized list Qs of length N. The *safe* predicate sets up constraints which test the permutations as soon as they are generated by *permutation* predicate. Thus each queen is checked as it is being placed. Since two queens are not placed on the same row or column, the predicate need only check whether two queens attack each other along a diagonal. Predicate *safe* is defined recursively. A list of queens is safe if the queens represented by the tail of the list are safe, and the queen represented by the head of the list does not attack any of the other queens. The definition of *notattack(Q,Qs)* is a neat encapsulation of the interaction of diagonals. A queen is on the same diagonal as a second queen N columns away if the second queen's row number is N units greater than, or N units less than, the first queen's row number. The diagonals are tested iteratively until the end of the board is reached. The second clause for *no_attack* may be rewritten using *dif.*

The query *queens(4,Qs), write(Qs), nl, fail.* gives two solutions namely Qs = [2, 4, 1, 3] and Qs = [3, 1, 4, 2]. The query *queens(8,Qs).* gives the solution Qs=[1, 5, 8, 6, 3, 7, 2, 4]. There are however 92 solutions for 8-queen problem which can easily be generated by giving the proper query.

The main advantage of constrained terms and the coroutine mechanism is to apply tests as soon as possible. But the computation paradigm is still "generate-and-test": an enumeration procedure is used to generate values and the constraints are used in a "passive" way to test these values. Therefore the search space is reduced "a posteriori" after detecting failures due to inconsistent instantiation of variables.

4.2.2 Using constraint solvers

Intertwining generation and testing can also be done by adding the tests as constraints [17] before the generation. Thus instead of generating the complete set of permutations, many of which have no chance of being solutions, only some of the permutations are generated. Refer to Appendix F.3, the Light Meals problem and to Appendix F.4, the Cryptarithmetic Puzzle for illustrating the use of constraints in this manner.

Another way to use constraints is the "active" way which consists in reducing the search space in an "a priori" way by removing inconsistencies, i.e. combinations of values which cannot appear together in a solution. This approach requires the use of some advanced techniques like constraint propagation, consistency checking and sophisticated tree searching like lookahead procedures. Thus a programmer formulates the constraints of the problem at hand in a generate-and-test like manner, and the execution mechanism of the CLP-environment applies various efficient constraint satisfaction techniques to solve this set of constraints. Refer to [29] for further details on Finite Domain CLP. We have not implemented constraints in finite domains in the current system.

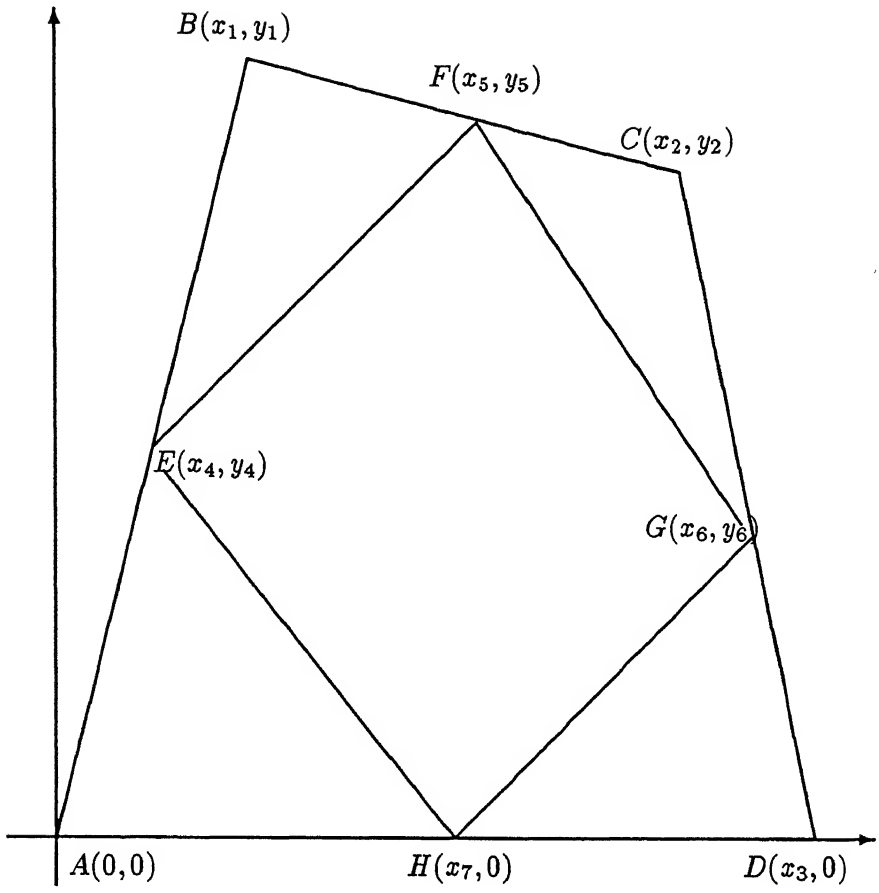


Figure 4.5: Geometrical Theorem Proving

4.3 Geometrical Theorem Proving

We consider the following theorem taken from [2] :
Let ABCD be an arbitrary quadrangle. Refer to Figure 4.5. Let E, F, G, and H be mid-points of edges AB, BC, CD, DA, respectively. Then the quadrangle EFGH is a parallelogram.

To prove the theorem, this geometrical problem can be transformed to an algebraic problem by introducing the Cartesian coordinate system. A midpoint (x_2, y_2) of a segment $(x_1, y_1)-(x_3, y_3)$ can be represented by equations $x_2 = (x_1+x_3)/2, y_2 = (y_1 + y_3)/2$. The fact that segment $(x_1, y_1) - (x_2, y_2)$ is parallel with the segment $(x_3, y_3) - (x_4, y_4)$ can be represented by an equation :

$(y_2 - y_1)/(x_2 - x_1) = (y_4 - y_3)/(x_4 - x_3)$.

These equations are represented as follows :

```
mid(X1,Y1,X2,Y2,X3,Y3) :-
   {2.0 * X2 = X1 + X3, 2.0 * Y2 = Y1 + Y3}.
para(X1,Y1,X2,Y2,X3,Y3,X4,Y4) :-
```

```
{(X1-X2) * (Y3-Y4) = (Y1 - Y2) * (X3 - X4)}.
```

By evaluating the following query against the above program, the given theorem is proven.

```
goal :-
   mid(0.0, 0.0, X4, Y4, X1, Y1),
   mid(X1, Y1, X5, Y5, X2, Y2),
   mid(X2, Y2, X6, Y6, X3, 0.0),
   mid(X3, 0.0, X7, 0.0, 0.0, 0.0),
   para(X4, Y4, X5, Y5, X7, 0.0, X6, Y6),
   para(X4, Y4, X7, 0.0, X5, Y5, X6, Y6).
```

The basic idea of this program and query is to certify that two pairs of segments, whose endpoints are constrained to be the midpoints of the original quadrilateral, are parallel.

Chapter 5

Vienna Abstract Machine

The Vienna Abstract Machine [23] is a Prolog machine developed at Technische Universität Wien. An inference in VAM (in contrast to the standard implementation technique, the Warren Abstract Machine - WAM [32, 22]) is performed by unifying the goal and the head immediately instead of by passing arguments through a register interface. We are using the VAM_{2P} [23] implementation since it is well suited for an intermediate code emulator. During an inference, VAM_{2P} fetches one instruction from the goal code and one instruction from the head code and executes the combined instruction. VAM performs cheap shallow backtracking, needs less dereferencing and trailing and implements a faster cut. Comparison of VAM with WAM is given in Appendix B.

5.1 Representation of clauses

The representation of clauses in VAM intermediate code is very close to their syntactic representation. The three different kinds of codes used are :

control codes Control Codes are used to embrace goals. A goal starts with c_goal < > and either ends with a c_call if another goal follows or ends with c_lastcall if it is the last goal in a clause. c_cut is used for the cut choice points.

head codes Head Codes are used to encode terms in the arguments of a clauses head. The arguments are translated into flat prefix code. The following head codes are used :

h_nil, h_const, h_list, h_struct, h_void, h_fstvar,
h_nxtvar, h_fsttmp and h_nxttmp.

goal codes Goal Codes encode terms in goals. The structure is the same as for head codes. The following goal codes are used :

g_nil, g_const, g_list, g_struct, g_void, g_fstvar,
g_nxtvar, g_fstuns and g_nxtuns.

5.2 Memory model

The representation of dynamic terms is based on structure copying. The VAM uses three stacks :

1. The Environment Stack contains the stack frames which hold the local variables and control information. It is either a determinate or a nondeterminate stack frame (choice point).
2. Lists and structures are stored on the Copy Stack.
3. Variable bindings are stored on the Trail.

In the current implementation all this except backtracking in case of cuts is taken care of by Prolog itself. Thus these three stacks are neither specified nor used directly. Instead only a list of choice points is passed for backtracking.

5.3 Specification of VAM

The specification describes the process of unification and (determinate) control in detail. Backtracking aspects and cuts are not explicitly covered. Backtracking in absence of cuts is implicit in the specification.

5.3.1 An Abstract Interpreter for VAM

The interpreter consists of a tail recursive predicate vam_prove/3 which holds the interpreter state consisting of the remaining head goals, the list of remaining goal codes, a continuation stack for nested calls. 'vam_prove/3' has the form :

```
% vam_prove(HeadCode,GoalCode,Stack)
% vam_prove/3 : Check and execute the instructons at the two
%               instruction pointers after combining them.
   vam_prove([c_nogoal],[c_lastcall],[]).
   vam_prove([H|Hs],[G|Gs],St) :-
      unification(H+G,Hs,NHs,Gs,NGs),
      vam_prove(NHs,NGs,St).
   vam_prove([H|Hs],[G|Gs],St) :-
      resolution(H+G,Hs,Gs,NGs,St,NSt,NextPred),
      vam_clause([NextPred|NHs]),
      vam_prove(NHs,NGs,NSt).
```

The details of cuts have been omitted here for simplification. This has however been explained in a later section on extensions of VAM. The process of interpretation consists of unification and resolution. These steps correspond to the different kinds of codes. The state transitions are specified by facts in order to emphasise which states are changed. Before trying to prove these facts, the interpreter takes the first elements of both lists (head and goal code) and combines them in order to pass

them to the facts. The effective instructions $HeadCode + GoalCode$ are derived by generating all valid combinations of head and goal codes.

5.3.2 Unification

An attempt is made to unify corresponding arguments of head and goal; the remaining codes are passed back to vam_prove/3. Unification/5 has the form :

unification(Instruction, HeadsIn, HeadsOut, GoalsIn, GoalsOut)

where Instruction is Head + Goal. In the actual implementation, a value called UnificationIndex is used for Head + Goal as shown in vam_prove/7. Combinations such as *h_struct* + *g_const* are not stated; they simply fail. Unification/5 only changes the head and goal code lists. Arbitrarily nested structures occuring in the head and the goal need neither a push down stack nor a counter to unify their arguments since the functors F/A do not insert their arity into vam_prove/3's state. Some of the combinations are illustrated below :

```
% unification(Instruction, HeadsIn, HeadsOut, GoalsIn, GoalsOut)
% unification/5 : Unify corresponding arguments in HeadsIn and
%                 GoalsIn depending on Instruction and return the
%                 remaining code via HeadsOut and GoalsOut.
   unification(void+void,Hs,Hs,Gs,Gs).
   unification(void+fstvar,Hs,Hs,[var(_)|Gs],Gs).
            . . .
   unification(fstvar+fstvar,[var(V)|Hs],Hs,[var(V)|Gs],Gs).
   unification(fstvar+nxtvar,[var(V)|Hs],Hs,[var(V)|Gs],Gs).
            . . .
   unification(constant+fstvar,
               [C|Hs],NewHs,[var(V)|Gs],NewGs) :-
      nonvar(V), !, V = [V1|VRem],
      unification(constant+V1,[C|Hs],NewHs,VRem,TempGs),
      append(TempGs,Gs,NewGs).
   unification(constant+fstvar,
               [C|Hs],Hs,[var([constant,C])|Gs],Gs).
            . . .
   unification(constant+constant,
               [const(Const)|Hs],Hs,[const(Const)|Gs],Gs).
            . . .
   unification(struct+void,Hs,NHs,Gs,Gs) :-
      parse_dl(_,[struct|Hs],NHs).
            . . .
   unification(struct+nxtvar,Hs,NHs,[var(V)|Gs],NewGs) :-
      nonvar(V), !, append(V,Gs,TempV),TempV=[V1|V2],
      unification_index(struct,V1,UIndex),
      unification(UIndex,Hs,NHs,V2,NewGs).
   unification(struct+nxtvar,Hs,NHs,[var(V)|Gs],Gs) :-
```

```
    parse_dl(V,[struct|Hs],NHs).
unification(struct+struct,Hs,Hs,Gs,Gs).
```

"Parse_dl" has the form :

parse_dl(Variable, SymbolsIn, RemainingSymbols)

The function of 'parse_dl/3' is to unify 'Variable' with the first structure from 'SymbolsIn' and return the remaining symbols in 'RemainingSymbols'.

5.3.3 Resolution (Goal Selection)

A clause of NPred meaning NextPredicate is selected by the interpreter with the database predicate vam_clause/1. If a fact in the head was proved and if the body contains another subgoal (*c_nogoal* + *c_call*), the new goal is selected. The stack is not affected at all. If the head unifies and the goal was the last in the caller's clause (*c_goal* + *c_lastcall*), the head code will become the new goal code. Again the stack is not altered (last call optimization). If the head unifies and there is another goal in the caller's clause (*c_goal* + *c_call*), then the continuation is pushed into the stack, the head code becomes the new goal code and the interpreter switches to the new clause's head. The stack needs to be popped if a fact unifies and if the goal is the last in the caller's clause (*c_nogoal* + *c_lastcall*). Execution proceeds with the popped continuation. If the stack is empty, the interpreter halts successfully. Resolution/7 has the form :

```
% resolution(Head, Goal, Heads, GoalsIn, GoalsOut,
%            StackIn, StackOut, BTOPIn, BTOPOut, NextPred)
% resolution/5: VAM_Goal selection
   resolution(c_nogoal,c_call,[],[c_goal,c_cut|Gs],Gs,
         St,St,[_,BTOP|BT],[BTOP|BT],functor('!',0)) :- !,
      newcutbacktrack(BTOP).
   resolution(c_nogoal,c_call,[],
         [c_goal,v(4),functor(F,A)|Gs],Gs,
         St,St,[_|BT],BT,functor(F,A)) :- !.
   resolution(c_nogoal,c_call,[],
         [c_goal,v(3),const(atom(F))|Gs],Gs,
         St,St,[_|BT],BT,functor(F,0)) :- !.

   resolution(c_goal,c_lastcall,[c_cut|Hs],[],Hs,
         St,St,[BT1,_|BT],[BT1|BT],functor('!',0)) :- !,
      newcutbacktrack(BT1).
   resolution(c_goal,c_lastcall,
         [v(4),functor(F,A)|Hs],[],Hs,
         St,St,[BT1,_|BT],[BT1|BT],functor(F,A)) :- !.
   resolution(c_goal,c_lastcall,
         [v(3),const(atom(F))|Hs],[],Hs,
         St,St,[BT1,_|BT],[BT1|BT],functor(F,0)) :- !.
```

```
resolution(c_goal,c_call,[c_cut|Hs],Gs,Hs,
      St,[Gs|St],[BTOP|BT],[BTOP|BT],functor('!',0)) :- !,
   newcutbacktrack(BTOP).
resolution(c_goal,c_call,[v(4),functor(F,A)|Hs],Gs,Hs,
      St,[Gs|St],BT,BT,functor(F,A)) :- !.
resolution(c_goal,c_call,[v(3),const(atom(F))|Hs],Gs,Hs,
      St,[Gs|St],BT,BT,functor(F,0)) :- !.

resolution(c_nogoal,c_lastcall,[],[],Gs,
      [[c_goal,c_cut|Gs]|St],St,
      [_,_,BTOP|BT],[BTOP|BT],functor('!',0)) :- !,
   newcutbacktrack(BTOP).
resolution(c_nogoal,c_lastcall,[],[],Gs,
      [[c_goal,v(4),functor(F,A)|Gs]|St],St,
      [_,_|BT],BT,functor(F,A)) :- !.
resolution(c_nogoal,c_lastcall,[],[],Gs,
      [[c_goal,v(3),const(atom(F))|Gs]|St],St,
      [_,_|BT],BT,functor(F,0)) :- !.
```

The 'newcutbacktrack' call marks all choice points from the one stated to the current one as unbacktrackable. This disallows the choice points from activating their choices from the point at which the requisite 'newgetbacktrack' call was made when backtracking takes place.

Chapter 6

Structure Independent Inference Engine

6.1 Introduction

The Vienna Abstract Machine has been modified and extended to handle constraints and it forms the underlying abstract machine for our inference engine. While the Constraint Solver only answers the satisfiability questions, the Inference Engine uses this capability to direct the collection of constraints. The inference engine thus provides a control mechanism and effectively executes derivations. Because of the inclusion of unification within the inference engine, the CLP system engine implements almost complete Prolog. The details of the internally provided predicates are given in Appendix A. Programs may thus be written in normal Prolog form. For the programs to differentiate between programmed predicates and constraint relation symbols, the constraints must be surrounded by braces as in Prolog-III. Consider an example taken from [19],

```
zmult(c(R1,I1),c(R2,I2),c(R3,I3)) :-
   {[float, [R3=R1*R2-I1*I2,I3=R1*I2+R2*I1]]}.
```

This single rule program models the relationship between two complex numbers and their products, where each complex number, X+iY, is represented as c(X,Y). The above rule thus means that multiplication of a complex number c(R1,I1) and c(R2,I2) equals c(R3,I3). The two equations in braces in the body of the above rule are constraints. 'float' specifies that the domain of these constraints is of type float. Other details of the constraint solver could be specified here before the list of constraints is specified in a list. The goal :

```
:- zmult(c(R,I),c(10.0,50.0),c(20.0,50.0))
```

entails the solution of the simultaneous equations :

```
20.0 = R * 10.0 - I * 50.0
50.0 = R * 50.0 + 10.0 * I
```

giving the result R = 1.038461 and I = -0.192307. An under-specified goal will result in a symbolic answer.

The same operators may be used both for unification as well as in constraints, the interface takes care of correct transmission of grounded terms between the constraint solver and the inference engine. In the above example, the equality symbol in the constraints indicates that it is an equality constraint and not the equal of Prolog which means unification. The constraints themselves may be specified in prefix form as in Prolog to make the specification structure independent or in infix form. For the latter, the "op" predicate will specify the precedence and arity (unary/ binary) of the operators used in the constraints as well as in normal Prolog. Currently the "op" predicate has not been implemented. Instead precedences for the common arithmetic operators are hardcoded.

6.2 Instruction set extensions

Since the constraint solvers will need a variety of domains, different types of constants like atom, str, float, int have been allowed. For this purpose, the instruction "constant" has been modified. The unification instructions have also been extended to handle the unification of these constants.

In the current implementation, new types of Domains like molecules [30], chemical graphs [3], or finite domains [29], etc. have to be specified as prolog terms or using the basic domain provided by the system. They cannot be specified and viewed in their natural form. This requires an extension of the instruction set and the unification instructions along with facilities for specification and a proper display of objects in these new domains.

6.3 Extensions to Model

To allow freezing of a goal and handling of constraints, three extra parameters have been added to the VAM interpreter - FrozenGoals, Constraints and the Dictionary. The goals which have been frozen because some variable is unbound are stored on the FrozenGoals Stack. As soon as a frozen variable gets bound, all goals waiting on that variable are popped from the FrozenGoals Stack one by one and executed. The constraints which are to be passed to the constraint solver are stored on the constraints stack. The variables used by the constraints are stored in the Dictionary with their corresponding counter-parts in the solvers. The predicate thawed_vam_prove/8 is used to activate the unfrozen goals. The new vam_prove/7 and thawed_vam_prove/8 are shown below :

```
% vam_prove(Heads, Goals, St, BTOPList, FrozenGoals,
%           Constraints, Dictionary)
% vam_prove/7: Abstract interpreter
   vam_prove([c_nogoal],[c_lastcall],[],_,FrozenGoals,
         Constraints,Dictionary) :-
```

```
   write('Frozen Goals:'),
   user_understandable_write(FrozenGoals,Dictionary),nl,
   write('Constraints :'),
   user_understandable_write(Constraints,Dictionary),nl,!,
   assign_constraint_values(Dictionary).
vam_prove([H|Hs],[G|Gs],St,BTOPList,FrozenGoals,
      Constraints,Dictionary) :-
   write_if_debug('Head:',H,', Goal:',G),
   unification_index(H,G,UIndex),
   unification(UIndex,Hs,NHs,Gs,NGs),!,
   vam_prove(NHs,NGs,St,BTOPList,FrozenGoals,Constraints,
      Dictionary).
vam_prove([H|Hs],Gs,St,BTOP,FrozenGoals,
      Constraints,Dictionary) :-
   check_and_solve_constraint(H,Constraints,NewConstraints,
      Dictionary,NewDictionary),
   thawed_vam_prove([H|Hs],Gs,St,BTOP,[],FrozenGoals,
                    NewConstraints,NewDictionary).

thawed_vam_prove([H|Hs],[G|Gs],St,BTOP,
      UnsolvedFrozenGoals,[],Constraints,Dictionary) :-
   resolution(H,G,Hs,Gs,NGs,St,NSt,BTOP,NBTOP,NPred),!,
   write_if_debug('Solving ',NPred),
   solve_predicate(NGs,NSt,NBTOP,NPred,UnsolvedFrozenGoals,
      Constraints,Dictionary).
thawed_vam_prove(Hs,Gs,St,BTOP,
      UnsolvedFrozenGoals,[[X|Goal]|FrozenGoals],
      Constraints,Dictionary) :-
   var(X),!,
   append(UnsolvedFrozenGoals,[[X|Goal]],
      NewUnsolvedFrozenGoals),
   thawed_vam_prove(Hs,Gs,St,BTOP,
         NewUnsolvedFrozenGoals,FrozenGoals,
         Constraints,Dictionary).
thawed_vam_prove(Hs,Gs,St,BTOP,
      UnsolvedFrozenGoals,[[_|Goal]|FrozenGoals],
      Constraints,Dictionary) :-
   convert_to_call(Goal,NVar),
   do_proper_append(Hs,NVar,NewHs),
   thawed_vam_prove(NewHs,Gs,St,BTOP,
         UnsolvedFrozenGoals,FrozenGoals,
         Constraints,Dictionary).

do_proper_append([c_nogoal],TempHs,NewHs) :-
```

```
    append([c_goal|TempHs],[c_lastcall],NewHs).
do_proper_append([c_goal|Rest],TempHs,NewHs) :-
    append([c_goal|TempHs],[c_call,c_goal|Rest],NewHs).
```

Examples of the VAM code which is executed by the above routines are given in Appendix E.

Chapter 7

Compilation

In the current implementation, the user queries and the clauses in a user program are compiled separately. Each compiler which is written in Prolog and C (LEX and YACC) respectively is explained below.

7.1 Compiler in Prolog

The compiler for compiling user queries is implemented in Prolog. For this we have used the Operator Precedence Parsing [1]. The form of the compiler is :

```
compile(Characters,VAMCode) :-
    tokenize(Characters,Tokens),
    parse(Tokens,IntermediateCode),
    encode(IntermediateCode,VAMCode).
```

The scanner breaks a STRING up into a list of tokens of the following kinds:

```
TOK     = lbrack; rbrack;     [ ]
          lpar; rpar;         ( )
          lcurly; rcurly;     { }
          var(STRING);        X,  This, _ok
          atom(STRING);       hello, :-, :::, fail

          int(INTEGER);       123, -456
          real(REAL);         3.50e-2, 5.79
          str(STRING);        "This is a STRING"
          char(CHAR);         '*

          comma;              ,
          bar;                |
          dot                 .
```

A symbol is either a name starting with a lowercase letter, or a sequence of following characters:

```
+  -  *  /  =  `  :  .  \  ^  <  >  ?  @  #  $  &
```

The associativity and precedences defined for operators are :

```
op_db(1200,xfx,(':-')).        op_db(1200,fy,(':-')).
op_db(1100,xfy,';').           op_db(1000,xfy,',').
op_db(900,fy,'not').
op_db(700,xfx,'=').            op_db(700,xfx,'\=').
op_db(700,xfx,'is').           op_db(700,xfx,'<').
op_db(700,xfx,'=<').           op_db(700,xfx,'>').
op_db(700,xfx,'>=').           op_db(700,xfx,'==').
op_db(700,xfx,'\==').          op_db(700,xfx,'=..').
op_db(500,yfx,'+').            op_db(500,fx,'+').
op_db(500,yfx,'-').            op_db(500,fx,'-').
op_db(400,yfx,'*').            op_db(400,yfx,'/').
op_db(300,xfx,'mod').
```

Clauses could also be compiled using the above compiler as illustrated below. Consider the program for append :

```
append([],Rest,Rest) :- !.
append([Ch|Rest1],Rest2,[Ch|NewRest]) :-
   append(Rest1,Rest2,NewRest).
```

The output for the text consisting of the above append program is :

```
[t(atom(append),0),t(lpar,6),t(lbrack,7),t(rbrack,8),
 t(comma,9),t(var(Rest),10),t(comma,14),t(var(Rest),15),
 t(rpar,19),t(atom((:-)),21),t(atom(!),24),t(dot,25),
 t(atom(append),27),t(lpar,33),t(lbrack,34),t(var(Ch),35),
 t(bar,37),t(var(Rest1),38),t(rbrack,43),t(comma,44),
 t(var(Rest2),45),t(comma,50),t(lbrack,51),t(var(Ch),52),
 t(bar,54),t(var(NewRest),55),t(rbrack,62),t(rpar,63),
 t(atom((:-)),65),t(atom(append),71),t(lpar,77),
 t(var(Rest1),78),t(comma,83),t(var(Rest2),84),
 t(comma,89),t(var(NewRest),90),t(rpar,97),t(dot,98)
]
```

The above scanned output when given to the parser, produces intermediate code as given below.

```
For first clause of append :
   cmp((:-),[cmp(append,[nill,var(Rest),var(Rest)]),atom(!)])

For second clause of append :
   cmp((:-),[cmp(append,[listparser(var('Ch'),var('Rest1')),
   var('Rest2'),listparser(var('Ch'),var('NewRest'))]),
   cmp(append,[var('Rest1'),var('Rest2'),var('NewRest')])])
```

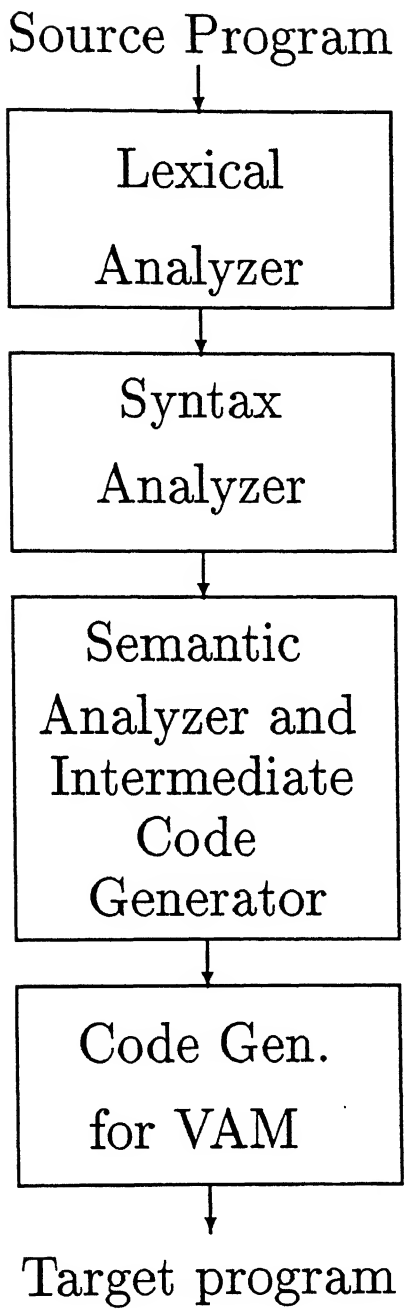


Figure 7.1: Phases of the compiler

The code generated for clauses is as follows :
For the first append clause we have :

```
VAMCode:
    [v(4),functor(append,3),v(3),const(nill),v(1),
     temporaryvar('Rest'),v(2),temporaryvar('Rest'),
     c_goal,c_cut,c_lastcall
    ],
Variables : ['Rest']
```

For the second append clause, the code generated is :

```
VAMCode:
    [v(4),functor(append,3),v(5),v(1),temporaryvar('Ch'),
     v(1),temporaryvar('Rest1'),v(1),temporaryvar('Rest2'),
     v(5),v(2),temporaryvar('Ch'),v(1),temporaryvar('NewRest'),
     c_goal,v(4),functor(append,3),v(2),temporaryvar('Rest1'),
     v(2),temporaryvar('Rest2'),v(2),temporaryvar('NewRest'),
     c_lastcall
    ],
Variables : ['NewRest','Rest2','Rest1','Ch']
```

As a general parsing technique, Operator Precedence Parsing has a number of disadvantages. It is hard to handle tokens like the minus sign which has two different precedences depending on whether it is unary or binary. Worse, since the relationship between a grammer for the language being parsed and the operator precedence parser itself is tenuous, one cannot always be sure the parser accepts exactly the desired langauge.

7.2 Compiler in C (LEX and YACC)

In the current implementation clauses are compiled using the compiler written in C (LEX and YACC). This is useful for large programs where code can be produced and stored in intermediate form which can be directly loaded by the simulator. The phases involved in the compiler have been shown in Figure 7.1. Note that optimization phase has not been implemented. The complete grammer used for YACC is shown in Appendix G.

The VAM-code for a program using constraints, namely the installments capital problem [9, 10] is shown in Appendix E.1 and a section of the code for Example 13 in Appendix C.9 that uses freeze [5] in Appendix E.2.

Chapter 8

Solvers

The real/float solvers currently implemented delay consideration of any nonlinear arithmetic constraints and linear inequalities. These constraints are assumed to be satisfiable until they become linear at some later time in a computation (this may happen when variables contained in a non-linear constraint or all variables in linear inequalities become ground), at which stage it will be considered for satisfiability. Thus it represents a compromise between generality and efficiency. It restricts consideration to a class of constraints which can be solved efficiently. However, by relaxing the satisfiability condition for such constraints rather than disallowing them altogether, the solver admits an important additional class of programs.

Linear arithmetic constraints (equalities) are maintained in solved form. As each linear constraint is added, it is checked for satisfiability with the previous constraints (already in solved form). If the system is satisfiable, the new constraint is incorporated to generate a new solved form. A delayed constraint is added to the solved form when a sufficient number of its variables have been determined to make it linear. For example,

```
hyp(X,Y,pow(X*X + Y*Y, 0.5)).
available(X,Y) :- {[real,[X = Y]]}.

goal :- {[real,[X + Y = 10.0, Z =< 8]]},
        hyp(X,Y,Z), available(X,Y).
```

The execution proceeds by collecting the two linear constraints, $X - Y = 10.0$, and $Z =< 8$. Next, the matching of the goal atom hyp(X,Y,Z) and the head of the rule hyp(X,Y,pow(X*X+Y*Y,0.5)) gives Z = pow(X*X+Y*Y,0.5). This constraint is delayed since it is not linear. The next constraint collected is X = Y, and this together with $X + Y = 10.0$, determine the values X = 5.0 and Y = 5.0. Now the delayed constraint is used and Z = pow(5*5 + 5*5, 0.5) = 7.07 is added to the linear constraints. Since the linear constraints are satisfiable, the goal succeeds.

We classify the solvers basically into two types – the black_box solvers and the incremental solvers.

8.1 Black Box Solvers

A black_box solver is assumed to take in a system of constraints and return a simplified system of equivalent constraints. If the system of constraints is satisfiable, a 'true' status is returned. If the set of constraints is not satisfiable, then the solver returns 'fail' indicating that the inference engine should backtrack. An 'error' status is returned in case any error occurs in the interpretation of constraints or in case of communication errors. In this type of solver, each time a constraint is added or any of the constraint variables are unified with a term by the inference engine the solver is called with the whole set of collected constraints. Also, before the call the current state of the constraints has to be stored.

8.2 Incremental solvers

In the case of incremental solvers a single constraint is added to the set of constraints already stored at the constraint solver. The incremental solvers may or may not themselves backtrack.

- In the case of incremental solvers, which do not provide backtracking to the earlier state of the solver in case of failure, storing of the current state is imperative before a constraint is given to the solver so that backtracking can be simulated by initializing the solver and then sending the current state to the solver.

- If the solver can backtrack, then a specific call must be given to the solver by the inference engine when we want it to backtrack because of some condition external to the solver. However, if the addition of the current constraint makes the system inconsistent, then the solver itself backtracks to its earlier state and returns a status 'fail' to the Prolog Inference Engine allowing the inference engine to take proper action. In this case, no backtrack call is given to the solver, since it has already backtracked.

  Special care has been taken in our implementation to send backtrack calls even in case of cuts. Thus if a cut is present after a set of constraints was satisfied, and a failure later does not allow the state to backtrack to the earlier set of constraints (because of the cut), still proper backtrack calls will be sent to the incremental solver, ignoring the cut, so that synchronization is maintained between the inference engine and the incremental solver. Refer to Appendix D.4 for further details with the help of examples.

In incremental solvers, in contrast to the Black Box solvers, the full set of constraints need not be given to the constraint solver, instead only the latest constraint is added to the set of constraints already stored by the solver. This minimizes the overhead of communication between the solver and the inference engine. However, no values are directly returned. Only consistency is checked for. Thus whenever any outputs are desired, a special 'output' call needs to be given to the solver with the variables

names for which values are wanted. Thus these calls incur an extra cost over the black box solvers.

8.3 Alternative Solutions

It is possible that the solvers may return multiple solutions. However this has not been currently implemented. The additional work that will be need to be done is to store the set of returned possibilities in the form of choice points. Each of the alternatives can be tried out later one by one as required once the inference engine has the control.

If the incremental solvers have multiple solutions, then the other possibility is that the solver will go through one set of solutions, but a backtrack call will allow the solver to try out the other possibilities automatically. This case has also not been implemented in this version.

8.4 Using pre-existing and custom built solvers

Many pre-existing solvers exist for specific domains. With a view to accomodate and take advantage of these pre-existing solvers (example Mathematica [33], REDUCE [14]), a very general interface has been provided between the structure independent engine and the constraint solvers. Facilities have also been provided to take advantage of incremental custom-built solvers. The solvers may be either on the host machine itself running the inference engine or on any other machine which can be accessed over the network. In the former case, the solver may be written directly by the user as an user program or it may be an independent solver attached as a module through the interface. In the latter case, currently 'rpc' (the remote procedure calls) is being used as the major method for communication. If the file system being used is common to the host machine and a solver, then the 'same_fs' method may also be used for communication. The technique of interfacing solvers to the Inference Engine has been elaborated in the next Chapter.

8.5 Invocation points for solvers

The CLP scheme specifies that the collected constraint set at each step must be satisfiable. However we have relaxed this condition and in the current implementation the solver is being automatically called only at the ends of derivations. If this becomes expensive, then solver_off/0, solver_on/0, solver_now/0 can be used. Thus the user can place satisfiability checks at key points in the program. Solvers for very complex domains may take a long time to solve even a small problem. Incorporating such solvers into CLP systems is impractical unless we can turn the solver off.

- *solver_off* stops the automatic invocation of the constraint solver.

- *solver_on* restarts the automatic invocation of the constraint solver at ends of derivations.

- *solver_now* invokes the constraint solver with the currently accumulated set of constraints right away. This is required to invoke the solver if automatic invocation has been put off.

Thus these three routines help in maintaining complete control over the invocation of the required constraint solver.

Chapter 9

Constraint-Solver Server

The interconnection of computers serving as a Constraint-Solver Server supporting our constraint logic programming system is described below. A computer on which the inference engine is running, makes available a set of constraints to other computers where constraint solvers are present, over a network shown as *Communication interface* in Figure 9.1.

- When a request occurs from one computer, which is called the client, it converts the constraints from the Inference Engine format to an Intermediate format, using a Filter and transmits them to another computer, which is called the server. The *Inference Engine*, the *Interface* and the $Filter_{I.E.}$ are all present on the client itself as shown in Figure 9.1.

- A Filter on a server machine converts the constraints from the intermediate form to the constrant solver form and passes the constraints to the constraint solver for evaluation. A number of servers containing a $Filter_{C.S.}$ and a constraint solver are shown in Figure 9.1.

- The constraint solver evaluates the constraints and returns the results (if any) to the client, again making the necessary transformation to the intermediate format for transmission over the network.

- The results in the intermediate language are converted to Prolog terms and VAM form for use by the inference engine.

Although, only a single solver is shown at each client, actually a number of solvers may be present along with their independent filters at each server. Solvers could be present on the client also, in which case requests need not be made over the network. The connection of such a solver is shown in Figure 9.1. Also note that the communication interface is not necessarily 'rpc', but could be any other method by which the inference engine and solver can interact, e.g. via files in case of Same file system.

Multiple solvers can be interfaced. The user only needs to specify the domain over which the constraints are based in his application. The system has to be

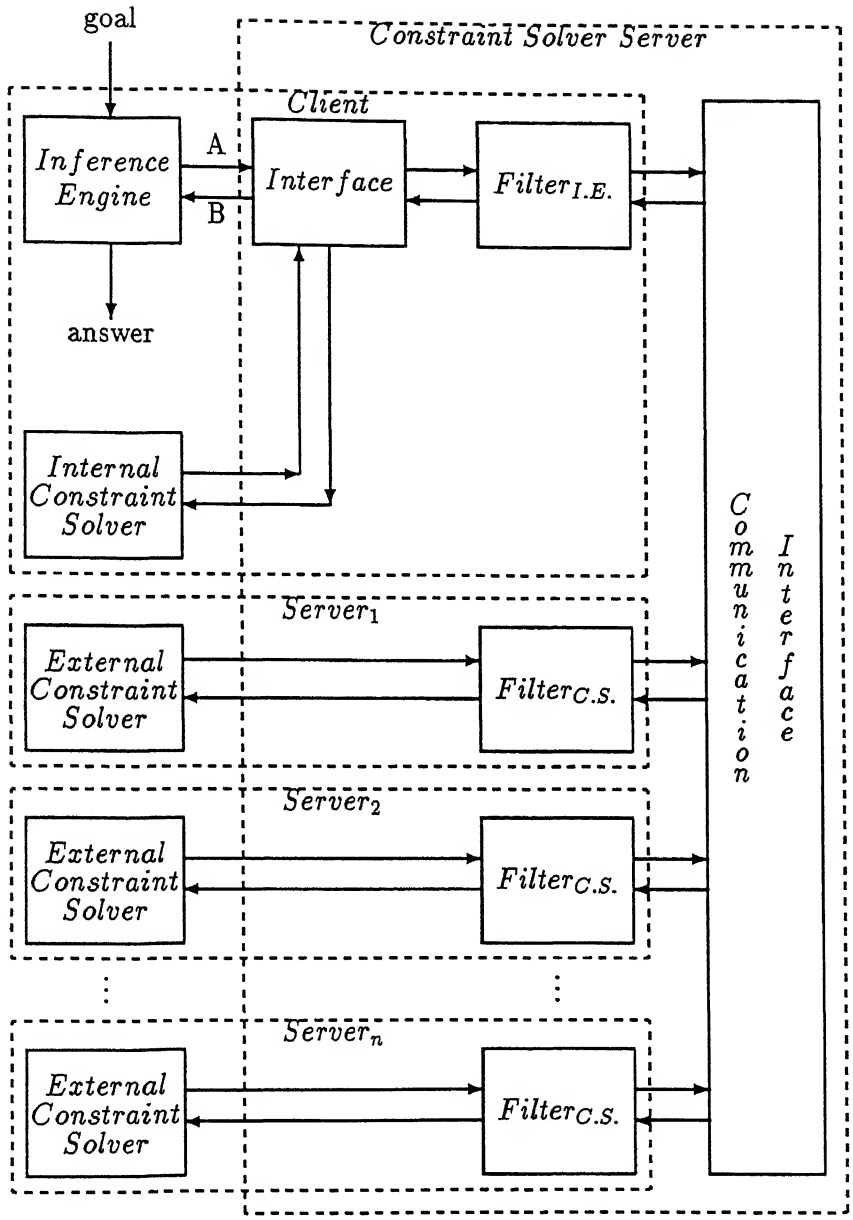


Figure 9.1: A diagram of a modular CLP system which contains the Prolog-like inference engine, a number of constraint solvers distributed over the network, and the Constraint-Solver Server constituting the interface and filter on the client side and a number of filters connected to their respective constraint solvers at the servers.

detailed through the advice database about the available constraint solvers. The advice database is maintained by the system administrator. This allows the interface to select the proper constraint solver for invocation.

9.1 Advice Database

The advice database has been implemented as facts in Prolog which specify the details of different kinds of solvers available over the network. Each fact has the following fields : DomainOfSolver, Strategy, Filter, TypeOfSolver, SolverName, OtherDetails

The first field is the *DomainOfSolver*. This could be any one of the domains selected from amongst float, char, string, list and int which are the basic domains currently provided by the inference engine. These domains could be used by the solver to solve constraints over other complex domains. See the example on multiplication of complex numbers in the Introduction Section of Chapter 6.

The *SolverName* is the name by which the Solver is referred to on the machine on which it is present. There could be a number of different types of solvers for the same domain residing on the same machine but having different names.

The *RemoteMachineName* is the name by which the remote machine is referenced. This name is required to make contact with the machine on which the desired solver is present. The name 'self' is used to indicate that the client and the server machines are one and the same.

The *Strategy* indicates the way the communication between the interface and the constraint solver is implemented. It could be either the RPC (Remote Procedure Call), or qprolog (the way Quintus Prolog allows invocation of a copy on another machine), or through files if the file system is shared between multiple machines or any other way the operating system might provide.

The *Filter* specifies the form in which the text is transmitted over the network from the client side. In the current implementation all constraints are transmitted in text form. Thus the constrants have to be converted from the VAM instructions to text-atoms with the help of a filter on the client side before transmission. For efficiency, the filter could be bypassed completely if the client and server machines are binary compatible. In this case the structures may be passed directly.

The *TypeOfSolver* field indicates whether the solver is a black-box solver. or an incremental one. This field is used to check the necessity to store the current set of constraints on the constraints stack. Also control of backtracking is done with the use of this field. This issue is dealt with in a later section. Examples of programs using black_box and incremental solvers are given in Appendix D.

The *OtherDetails* field contains rest of the details not indicated above specific to the communication strategy used. For usage of remote procedure calls, it contains the program number, version number, time-out etc. For same file system it contains the file number and time out. Any comments can be included in this field.

9.2 Interface

The CLP implementer can choose the most efficient internal representation for the constraint set, and is allowed to implement and interface the most suitable and efficient incremental constraint solving techniques. The interface determines the details for the specified domain from the Advice Database. The finding out of these details can be overridden if the user himself specifies the details completely. The constraints specified by the user in his application are converted to the following form :

```
merge_constraints([[Details, Constraint_1],
                   ...,
                   [Details, Constraint_mn]])
```

which has been given by the user as :

```
{[Domain_1,[Constraint_1_1, ..., Constraint_1_n]],
 [Domain_2,[Constraint_2_1, ..., Constraint_2_n]],
                .  .  .
 [Domain_m,[Constraint_m_1, ..., Constraint_m_n]]
}
```

where Domain represents the domain over which the constraints are to be solved. The default Domain is 'float' and the default solver is 'solve_float_equation' if the constraints are specified without any details as shown below :

```
{Constraint_1, Constraint_2..., Constraint_n}
```

The users may specify their own Domains and Solvers after the required constraint solvers have been interfaced. The user may specify the full details instead of the Domain above as :

```
[real, rpc, machine_name=tejas, program_number=99,
 version_number=1]
```

where Domain is 'real', Strategy is 'rpc', the machine on which the solver is present is 'tejas' and further parameters are specific to remote procedure calls. Those fields which are left unspecified have default values.

9.2.1 Interface Routines for Internal Solver

The following routines have been defined for interfacing internal solvers to the inference engine.

A] convert_term_to_constraint_form This routine is used to convert the VAM terms specified in Prolog into an internal form suitable for manipulation by the Internal Constraint Solver. All Variables are replaced by distinct names for use by the constraint solver. This routine must be invoked before invocation of the constraint solver with a new constraint.

B] convert_constraint_to_term_form This routine which is the inverse of A] converts the terms from the internal form suitable for the constraint solver to VAM code required by the Prolog Inference Engine. This routine needs to be invoked only when symbolic outputs are desired. Till then the interface/solver will hold the constraints in unconverted form. The new set of collected constraints when required can be converted to VAM form with variables and output symbolically.

C] solve_constraints This routine is invoked whenever any constraint variable is changed or any new constraint is added. This invokes the proper constraint solver with the newly added constraint in internal form along with the earlier set of simplified constraints also in internal form which were retained on the Constraints Stack and determines whether the augmented Set is Consistent or Inconsistent with the existing set of constraints. If the solver itself is able to retain the earlier set of constraints (i.e. it is backtrackable), then no constraints need to be held on the Constraint Stack. Only fresh constraints need to be given to the solver in internal solver form. If the solver returns an output set of constraints, then the returned contraints are held in the inference engine in the Constraints stack.

D] output_constraints If the engine does not maintain the constraints but the solver does then the solver should provide a facility to show the current set of constraints directly and therefore this routine is separately required.

9.2.2 Interface Routines for External Solvers

For solvers on different machines, the internal form is to be converted to a form which the solvers can understand, and filters are used for this purpose. The same routines as above are used here also. Only the constraints are redirected to the filter if the solve_constraints call is for an external solver. The filter on the client side converts the internal form to a form which can be communicated over the interface. This format is fixed, so that new filters to be added on servers can all understand and use a single format.

9.2.3 Interface Structure

The Interface structure basically consists of a Dictionary currently implemented as a list of records (accessed sequentially) with each record of the form :

(Domain, PrologVariable, SolverVariable, PermanentName)

The Domain of a PrologVariable indicates the set of values which the PrologVariable may take. Thus even if the SolverVariable is used across solvers, the Domain of the PrologVariable does not change. The variable pair consisting of the 'PrologVariable' and the 'SolverVariable' are the variables in the Inference Engine and the Constraint Solver respectively. Whenever terms are sent to the solver, the variables are stored in a dictionary thus changing their status from variable to solver variable. This however in no way prevents unification of the solver variables by

the inference engine. Thus unification is allowed to be used in conjunction with constraint solving. Backtracking past the points where a variable was changed to a solver variable necessiates the changing back of the solver-variable status back to variable, thus removing the variable from the dictionary. Whenever a solver variable gets unified with a constant (atom, numeric or character), a new equation 'solver variable = constant' needs to be sent to the solver. If a functor gets unified with a solver variable we assume that the functor is comprehensible to the solver and again send an equation 'solver variable = functor(parameters)' (Refer pow/2 in hypotenuse example in beginning of Chapter Solvers) to the solver. If a normal prolog variable gets unified with a solver variable, we bind the solver variable to variable. If the solver variable unifies with another solver variable then an equation 'solver variable1 = solver variable2' is sent to the solver. These points have been explained further in later sections. Consider an example :

```
goal :- {X = Y + 3}, P(Y), write(X), nl.
p(8).
```

In the above case, Y is a solver variable which gets unified with 8. Thus an equality constraint 'Y=8' is sent to the solver. Note that we have used '='. Our implementation currently assumes that '=' is understood by the solver as equality either directly or through the interface. If a Variable is used across incremental solvers, and the Variable is instantiated, then this value needs to be communicated to all the incremental solvers using this Variable. This is not implemented. Also a stack is required to store incompletely solved constraints if an incremental backtracking solver is not available. This is provided by the inference engine. Details about backtracking for incremental solvers have been illustrated with examples in Appendix D.4.

9.2.4 Interface from Inference Engine to Solver

For passing constraints to a constraint solver, the constraints should first be converted to an internal form suitable for the constraint solver.

If a Prolog Variable already exists in the Dictionary, then its corresponding SolverVariable is used for conversion of the Term to an Internal Representation. Note that the SolverVariable may in turn be bound to a simplified/unsimplified solution and if so, this solution will be used for conversion. However, if the variable does not exist in the Dictionary, then a new entry is added to the Dictionary with a new SolverVariable which is used for conversion. This is all that needs to be done before calling an internal solver.

The constraint solver is invoked as soon as merging of constraints is done. The constraint solver will return with solutions attached to each of the SolverVariables. These solutions are directly suitable for reuse by the constraint solver since they are in internal form. However if the system of constraints is not satisfiable, then backtracking can be initiated here.

For every SolverVariable in the Dictionary, if the attached solution is minimal (e.g. a constant which cannot be further simplified), then it is converted to VAM-code and the converted code is substituted in the corresponding PrologVariable. Also, this record entry is removed from the Dictionary. If the SolverVariable is a Variable itself, then no solution has been computed for that variable and the entry is left as it is. Partial solutions are also retained without modification.

For External Solvers, first the same steps as above for Internal solvers are performed. Then during the conversion of each constraint to intermediate form for transmission over communication interface, the variables have to be noted to create the structure containing the Solver Variable and External Name for use over communication interface. This is called 'CorrespondingVariables'. The External name is selected dynamically for the whole set of constraints in case of black-box solvers each time constraints are sent to the solver while for incremental solvers the Permanent Name from the Dictionary is used. Names Q1, Q2, ..., Qn are used as Constraint Variable Names to be sent to black box solvers. For incremental solvers, the variables once allocated a name must necessarily use the same name for communicating with the Incremental constraint solver because it will understand only the variable names that were given to it earlier. If new names are given to the same variable, the incremental solver is not in a position to understand this and will therefore treat the variables as new variables. Therefore the field PermanentName is required in the Dictionary. The permanent names given are P0, P1, ... Pm.

9.2.5 Interface from Solver to Inference Engine

For External Solvers, if results are returned in terms of variable names Q0, Q1, etc, then for black-box solvers, all the variables can be resolved using the 'CorrespondingVariables' List created as above. The incremental solvers will return results in terms of P0, P1 etc in case the results contain variables. It is possible however that some of these names are not present in the 'CorrespondingVariables' because the incremental solvers will return names it had retained in its earlier calls. Therefore the Dictionary needs to be used instead of the 'CorrespondingVariables' to resolve the variables in case of Incremental solvers when results are returned because of explicit call to output_variables.

For Internal Variables, whenever partial/final solutions of Variables are to be output to the user, first the Dictionary is scanned for the presence of the Variable. If the variable is present in the Dictionary, then there are three options :

1. If the solution attached to the SolverVariable is minimal, then this minimal solution after conversion to VAM-code is used for printing out the Prolog Variable specified.

2. If the SolverVariable is itself a variable, then the corresponding Prolog Variable is printed out as Unbound.

3. If the solution attached to the SolverVariable is not minimal, then this solution after conversion to VAM-code such that any Internal Solver Variables in the

solution are replaced by the corresponding Prolog Variables is used for printing out the Prolog Variable specified.

If the Variable is not present in the Dictionary then it is not a variable used by the constraints and is directly output as a normal Prolog Unbound Variable. Note that any other structures produced by unification which are to be printed are already in VAM code form and therefore nothing special needs to be done to output them.

9.2.6 Unification of Variables used in Constraints

There are four possibilities to be considered :

1. A Prolog Variable in the Dictionary (i.e. used in Constraints) will get bound (instantiated) to a Term which can be converted to solver form.
2. A Prolog Variable in the Dictionary will get bound to another Prolog Variable already present in the Dictionary.
3. The Prolog Variable in the Dictionary will get bound to a Prolog Variable not present in the Dictionary in which case nothing needs to be done.
4. The Prolog Variable in the Dictionary will get bound to a term which cannot be converted to solver form in which case it is clearly a programmer error.

These possibilities are compared in Table 9.1. The first three possibilities are elaborated below :

Bound Prolog Variables Before going to the constraint solver one or more of the Prolog Variables may get bound either by unification or because of solutions from constraint solvers invoked earlier. This condition is handled as follows :

```
For every bound Prolog Variable (LHS) and its corresponding
     SolverVariable (RHS) in the Dictionary do
{  Convert the VAM-code form LHS to the SolverForm TempRHS;
   Add an equation RHS = TempRHS to the set of constraints;
   Remove this record from the Dictionary;
}
```

This is illustrated in the example below.


```
 No.       Program
(01) goal :- {Z1 = A+B, Z2 = A-B},
(02)         solve1(Z1),
(03)         solve2(Z2),
(04)         write(A),nl,write(B),nl.
(05) solve1(10.0).
(06) solve2(2.0).
```

Z1 and Z2 are Prolog Variables used in constraints (same is the case with A and B).
The converted constraints are as shown below :

```
[_Z1=_A+_B, _Z2=_A-_B]
```

The underscore before a variable name indicates that it is a Name substituted for the variable (i.e. a SolverVariable). After they have been merged in Line Number (01) above, the Dictionary will be :

```
[(float,Z1,_Z1,_),(float,Z2,_Z2,_),
 (float,A,_A,_),(float,B,_B,_)]
```

Now, in solve1(10.0) in Line Number (05), Z1 unifies with the VAM-code [v(3), const(real(10.0))] which is actually the real number 10.0 and since there is a c_nogoal for code produced for solve1 meaning that predicate solve1 has no subgoals, it has only a head which has already been unified, therefore the routine 'check_and-_solve_constraints' is called. This finds the list as :

```
[(float,[constant,const(real(10.0))],_Z1,_),
 (float,Z2,_Z2,_),(float,A,_A,_),(float,B,_B,_)]
```

An equation $_Z1 = 10.0$ is added as a constraint and passed on to the constraint solver and the first record is removed from the above Dictionary. A similar condition occurs at solve2 on line (06) when Z2 gets unified with [constant, const(real(2.0))] and the corresponding entry is removed from the Dictionary. Finally, this results in the output :

```
6.0
4.0
```

meaning that A is assigned a value of 6.0 and B a value of 4.0.

Duplicate Prolog Variables There is a possibility that two or more variables which were independently inserted into the list of constraints get unified later by the Inference Engine or by the solution of constraints by any of the constraint solvers. This causes duplicate entries of PrologVariables in the Dictionary. The corresponding SolverVariables (possibly attached to simplified constraint solutions), however will be different unless this duplication has been caused by the action of a constraint solver itself. To take care of this possibility before going to the constraint solver, a check for duplicate variables has to be made. Thus,

```
For every pair of records (R1,R2) where
   R1 is (Domain,PrologVariable,SolverVariable1,PermName1) and
   R2 is (Domain,PrologVariable,SolverVariable2,PermName2)
   with duplicate PrologVariable do
{  Add an equation SolverVariable1 = SolverVariable2
```

```
      to the set of constraints, over 'Domain'.
    Remove one of the records, either R1 or R2 from the
      Dictionary.
    (One of the records must be retained).
}
```

This is illustrated by the example below :

```
 No.       Program
(01) goal :- {Z1=A+B, C-B=A, A+C=2*B, Z2=A-B+C+1},
(02)         eq(Z1,Z2),
(03)         write(A),nl,write(B),nl,write(C),nl.
(04) eq(A,B) :- A=B.          % unification here
```

Initially on line number (01), the four constraints will be given to the constraint solver. Z1 and Z2 are separate variables. The constraints are converted to an internal form :

```
[_Z1=_A+_B, _C-_B=_A, _A+_C=2*_B, _Z2=_A-_B+_C+1]
```

The Dictionary is :

```
[(float,Z1,_Z1,_),(float,Z2,_Z2,_),
 (float,C,_C,_),(float,A,_A,_),(float,B,_B,_)]
```

The constraint solver will simplify the constraints resulting in a new Dictionary :

```
[(float,Z1,1.5*_B+ -1.0*_B+_B,_),
 (float,Z2,1.0+_B,_), (float,C,1.5*_B,_),
 (float,A,1.5*_B+ -1.0*_B,_), (float,B,_B,_)]
```

Later at line number (02), the invocation of predicate eq defined on line number (04) results in unification of Z1 and Z2. Thus while checking for duplicate variables, Variables 1 and 2 are found to be duplicate. The partial solutions are shown below :

```
Duplicate 1,2
1::PrologVariable:Z2
   SolverVariable:1.5*_B+ -1.0*_B+_B
2::PrologVariable:Z2
   SolverVariable:1.0+_B
```

Therefore a new constraint :

```
float, [1.0+_B=1.5*_B+ -1.0*_B+_B]
```

is created and added to the list of constraints to be sent to the constraint solver. Entry 2 marked above will be removed from the Dictionary. This finally results in the correct output shown below :

```
1.0
2.0
3.0
```

That is, A has value 1.0, B has value 2.0 and C has value 3.0.

Unifying a Prolog Variable used in Constraint with a normal Prolog Variable In this third case, we have used the approach of just unifying the two variables directly. However this approach could present a problem. We elaborate this possibility with an example shown below :

```
01:   goal :-
02:      freeze(C, freeze(D, C > D)),
03:      {A+B=7.0},
04:      C=A, D=B,   % Case 3 applies here
05:      {B-A=1.0},
06:      write(A),write(B),write(C),write(D).
```

It is expected that a failure occurs after the constraint on line 05 is solved. This is because we expect that C = A = 3.0 and D = B = 4.0. This implies that $C < D$ and not greater than as frozen on line 02. However on actual execution it is seen that *goal* completes execution and outputs the expected values. But there is still the unsolved frozen goal $C > D$. This happens because internally the variables C and D are still uninstantiated. It is only the replicas of the variables _C and _D used in constraints which have been instantiated. Therefore $C > D$ was never executed. However, the required values were output because only the write can directly access the Constraint Variables of the corresponding Prolog Variables.

This possibility can occur only if a program is specifically written for doing so. However in normal examples, this condition does not arise. Taking care of this possibility fully will present a lot of overhead in searching for every frozen Prolog Variable in the Dictionary to find whether its corresponding Constraint Variable (if found) is instantiated and thus take proper action.However a simple way out could be to assign values returned by the solver directly to the Prolog Variables and do away with the Solver Variables completely. If this is done, then *goal* in the above example will fail as expected. This was the approach we had followed in the meta interpreter as illustrated by the example in Figure 3.1. However the problem in this case is that a special call to *sum_of_products* to simplify the final solutions is required. This is because the solver returns incompletely simplified solutions (terms) which get unified with variables, although they have a chance of being further simplified. Since the variables get instantiated to unsimplified solutions because of the strategy of earliest substitution, the structures which are returned by the solver are the ones that are output. Clearly this is not desirable.

Another possibility is that only fully simplified solutions attached to the Constraint Variables are communicated to their corresponding Prolog Variables. Doing this will allow the *goal* to fail as expected. Also the outputs will present no problem because they are assigned only if they are completely simplified. This approach is also not sufficient, however, because in the above example, it is quite possible that instead of the values 3.0 and 4.0 we got by solving the set of constraints $\{A+B = 7.0$ *and* $A - B = 1.0\}$, we may get some terms containing variables for some different set of constraints, and these terms are not in minimal form. Also instead of the $C > D$, we may have a complicated frozen predicate. In this case also the above

Unification Possibility	Change in dictionary	Action
Bound Prolog Variable	Remove instantiated entry	Add new constraint converted(LHS)=RHS
Duplicate Prolog Variable	Remove one of the duplicate entries	Add new constraint RHS1=RHS2
Prolog Variable with normal Variable	-	Unify variables
Prolog Variable with incorrect Term	-	Programmer Error

Table 9.1: Possibilities in Unification

example will be solved incorrectly and this is not desired.

A final possibility is that if a Prolog Variable, say A, in the Dictionary is to be bound to a Prolog Variable B not present in the Dictionary, then instead of binding A to B, we communicate the current value of the Solver Variable _A corresponding to the Prolog Variable A to B. This approach will also allow the above *goal* to fail as desired. This approach is also not sufficient.

Since taking care of this third case completely implies searching the dictionary, and the dictionary could be quite large, we assume that freeze and constraints are not used in the same program in a way that could create problems, and do not implement this strategy.

Chapter 10

Adding a new constraint solver

Adding solvers to create a new system is a task simply of specifying the details of the solver to be added. A filter needs to be written to convert from the intermediate language to the form understandable by the solver being added and then to reconvert the results returned by the solver to the intermediate format. We show examples of adding three different solvers.

10.1 Boolean Black Box Solver

Suppose that a solver for boolean constraints is to be added. The boolean solver is available such that the inference engine and the solver can share the same file system. We use the basic type 'int' as the domain for boolean. The details of this boolean solver are specified in the advice database as :

get_detail(boolean,file_number,[],10).
get_detail(boolean,solver_kind,[],black_box).

All that is required is that the constraint solver along with its interface must be executing on the remote/same machine. Now the program which requires this constraint solver can execute directly. Thus the user can use a new constraint solver quite easily.
Consider the following example taken from [2]. Refer to Figure 10.1.

```
/* Cross circuit */
/* Problem is to prove that circuit is a cross circuit */
/* Ref : "CAL", Page 273 Proc. FGCS 1988 */
circuit(X,Y,A,B) :-
   { [boolean,[I4 = (~ X) or I3,
               I3 = X and Y,
               I5 = (~ Y) or I3,
               I8 = (~ I4) or I3,
               I9 = (~ I5) or I3,
               A = I4 and I11,
               I11 = I8 or I9,
```

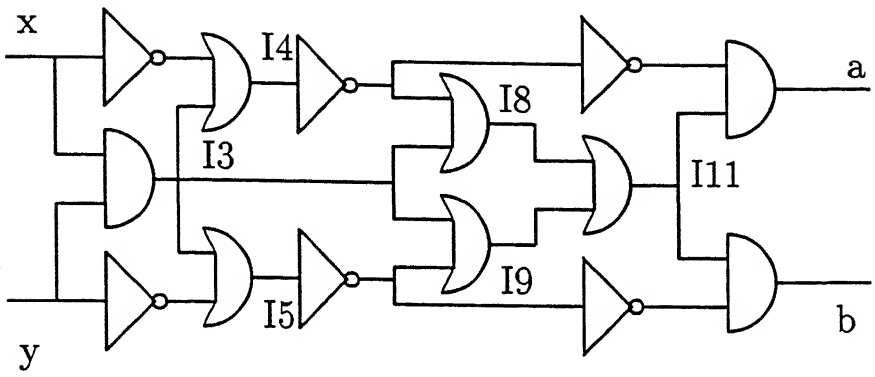


Figure 10.1: Cross Circuit

```
                B = I5 and I11
    ]          ]
  }.
goal :- circuit(X,Y,A,B),
   write('X='),write(X),nl,
   write('Y='),write(Y),nl,
   write('A='), write(A),nl,
   write('B='), write(B),nl.
```

In the above problem, the specification of the circuit has been described in terms of boolean equations. The relation between input and output terminals is also described. 'and' and 'or' are taken to be binary infix operators (functors) and '~' as a prefix unary operator (functor) for negation. When goal is executed, the resulting output is :

```
X=_X
Y=_Y
A=_Y
B=_X
```

where _X and _Y are variables. This is because, all the arguments in the query "circuit" were left free. The values which A, B, X, and Y can take are 'int' (used as boolean).

The domain 'boolean' specified in the above program is used to find details for the invocation of the requisite solver. A text filter is used by default. Refer to Example-103 and Example-105 in Appendix D for use of the boolean solver using the same common file system.

10.2 Real Black Box Solver

The details for a black box solver for domain real for which basic domain atom provided by the modified VAM is used, running on machine tejas with other details of 'rpc' is shown below :

```
get_detail(real,machine_name,[],tejas).
get_detail(real,constraint_prog,[],99).
get_detail(real,constraint_vers,[],1).
get_detail(real,function_number,[],1).
get_detail(real,tcp_udp,[],'tcp').
get_detail(real,solver_kind,[],black_box).
```

We can have a number of real solvers on tejas itself with different constraint_prog. We can change the name tejas to vayu if the real solver is available on machine vayu. As long as the solver is executing as a server on some machine and the interconnection network is up, all that needs to be done to attach a solver is specification of above details. Now the solvers can directly be used in the user program. Nothing more is necessary. Refer to Examples 101, 102, 105 in Appendix D for use of black box solvers in user programs using remote procedure calls ('rpc' mechanism).

10.3 Real Incremental Solver

The details of an incremental solver for domain inc_real for which basic domain atom provided by the modified VAM is used, running on machine tejas with other details of 'rpc' as actually used in the current implementation is shown below :

```
get_detail(inc_real,machine_name,[],tejas).
get_detail(inc_real,constraint_prog,[],97).
get_detail(inc_real,constraint_vers,[],1).
get_detail(inc_real,function_number,[],1).
get_detail(inc_real,tcp_udp,[],'tcp').
get_detail(inc_real,solver_kind,[],incremental).
```

Again in this case, nothing more needs to be done. Proper calls will be made to the incremental solver from the inference engine and outputs will be interpreted properly. Thus the incremental solver may be used easily by the user. Refer to Example-104 in Appendix D for use of an incremental solver for domain real.

Chapter 11

Conclusion and Future Work

11.1 Summary

We have presented a Modular Constraint Logic Programming System which allows the addition of different constraint solvers through the medium of an interface between the solvers and the inference engine. A solver is essentially a separate module and does not know anything about the inference engine. This allows selection of required solvers at run time. Fast prototyping of new CLP systems is thus possible. Solvers may run on different machines on the network. A Constraint Solver Server has been designed which allows the inference engine to access constraint solvers with the help of an advice database. Our system models the client server relationship. The inference engine forms the client and the constraint solvers are the servers which are remotely distributed. The solvers are contacted only when relevant constraints are to be solved. Thus the solvers can service a number of clients. Although the system in its current form is slow, we have demonstrated that it works for a number of real world problems. In this first level implementation we have not concentrated on efficiency but have successfully developed a prototype to illustrate the working of our design.

11.2 Current Implementation

A black box constraint solver using remote procedure calls for communicaton is implemented for domain Real on top of IF/Prolog on the HP-9000/850. This solver is invoked by the inference engine written in Quintus Prolog on the SUN-3 workstations. The HP-9000 and the SUN-3 workstations are connected through the Ethernet. We have also implemented the boolean and floating point constraint solvers on the SUN-3 which can also be invoked by the inference engine directly (Internal solvers). The boolean solver may be invoked using the same_fs strategy. An incremental solver for domain real has also been written on HP-9000/850 on IF/Prolog and it can also be invoked by the inference engine alone or in combination with other solvers.

Program	Solver	Time Taken
dc_circ1	External Real Black Box Solver	$61.150sec + 55.0sec.$
dc_circ1	Internal Float Black Box Solver	$44.817sec.$
dc_circ2	Internal Float Black Box Solver	$207.984sec.$
dirichlet1	Internal Float Black Box Solver	$25.334sec.$
dirichlet1	External Real Black Box Solver	$6.617sec + 55.0sec.$
dirichlet2	Internal Float Black Box Solver	$0.767sec.$
dirichlet2	External Real Black Box Solver	$0.800sec + 1.0sec.$
meals	Internal Black Box Float Solver	$3.084sec.$
meals	External Black Box Real Solver	$4.350 + 9.0sec.$
meals	External Incremental Real Solver	$5.167 + 13.0sec.$
mortgage	Internal Float Black Box Solver	$114.600sec$
mortgage	External Real Black Box Solver	$120.067sec + 160.0sec$
mortgage	External Incremental Real Solver	$120.783sec + 180.0sec$

Table 11.1: Execution Timings for some Constraint Logic Programs

The Vienna Abstract Machine has been extended to handle constraints. A number of large problems have been solved for testing the implementation. These include the Dirichlet problem for Laplace's equation for two dimensions, Electrical Engineering Problems [15], and Options trading Problem [19].

The practice of analyzing the complexity of Prolog Programs is not as developed as for programs in conventional programming languages. There has been little agreement on Prolog benchmarks other than LIPS or *logical inferences per second* which is probably the best measure. However we have noted the total time required for some of the programs from Appendix F to execute on the current implementation. These are shown in Table 11.1.

11.3 Applications

Constraint-based languages and systems have a demonstrated utility for a wide variety of applications including geometric layout, physical simulations, user interface design, document formatting, algorithm animation, design and analysis of mechanical devices and electrical circuits, and even jazz improvization [4]. Our system may be used to solve real world problems in a number of areas. Some applications in Scheduling and Planning as well as in Circuit Design are listed below :

Operations Research It consists of an inexhaustable source of interesting search problems, especially large scale scheduling and planning problems.

Disjuctive Scheduling Consider a problem in Civil Engineering. Problem is to minimize the total duration of building a bridge with precedence and disjunctive constraints due to the limited availability of resources.

Graph Colouring The problem is to find the minimum number of colours to label the vertices of a graph such that no two adjacent vertices are assigned to the same colour.

Car Sequencing This problem occurs in the scheduling for the assembly line of car manufacturing. Each car may require a different set of options and the assembly line has capacity constraints for the options. The problem is to generate a sequence of cars which satisfies the capacity constraints.

Optimal traffic assignment for satellites This problem is concerned with the scheduling of onboard switching systems in telecommunication satellites. The problem can be formulated as follows : given an interstation traffic matrix, determine the successive switching modes to switch all the traffic requirements in minimum time.

Warehouse location The problem is : given a set of warehouse locations, a set of customers, and the costs of stocking and transportation from warehouses to customers, to find an optimal configuration of the warehouses (i.e. their number and locations) which minimizes the total cost.

Cutting problems The problem consists in cutting two dimensional shelves of various sizes according to customer requirements from standard wood boards in a furniture factory. The objective is to minimize the total waste.

Investment planning The program chooses among different investment types in order to minimize or maximize a goal function over a given period. Using the symbolic simplex method, the program yields the most general solution and the user can then interact with it to get the best solution with regards to his need.

Circuit simulation Simulation of large combinatorial and sequential circuits is possible.

Symbolic Verification The formal comparison of an implementation of the circuit with its functional specification (usually a set of boolean equations) is possible.

Circuit Synthesis Problem is to automatically generate a circuit at the transistor level (in different technologies) from a truth-table or boolean equation specification.

Fault diagnosis The problem is to locate a faulty component in a circuit from its input/output misbehaviour.

Automatic test-pattern generation Problem is to generate a minimal number of test patterns to detect all single stuck-at errors in combinational circuits.

Circuit specialization and simplification The problem is : from a description of a circuit which performs a set of functions, to derive a more specialized

one performing only a subset of the functions. The goal is to minimize the number of components in the simplified circuit.

Channel routing This application comes from the area of VLSI layout design. It consists in connecting terminals on two sides of a rectangular channel in presence of certain constraints. The objective is to minimize the channel width.

Microcode label assignment This application comes from the area of computer firmware development. The problem is to assign labels of symbolic microcode to addresses in a page of microcode memory. Branch instructions generate constraints on certain bit patterns.

Some examples are included in Appendix F.

11.4 Future work

Our system is currently slow because the VAM is being simulated in Prolog. However if VAM_{1P} is used and code translated to machine code instead of simulating it, the speed will increase significantly. We have concentrated on designing a model, and not on optimization. Issues on optimization have been dealt with in [28].

Although new constraint solvers can be added, the representation of constraints is currently limited by the basic domains available in our implementation. An intermediate language must be designed which is the output of the filter from the inference engine side. This intermediate language can then be used by any of the filters on the solver side to convert it to a form understandable by the solvers.

A graphics interface needs to be added and the user should be allowed to specify his own domains in a proper format.

Specifications of constraint hierarchies may be added for constraint satisfaction which could include both the required constraints and the default constraints of differing strengths.

We are planning to further interface packages like Mathematica [33] and REDUCE [14] to our system. In the current implementation, we have restricted our attention to synchronous requests for solving constraints in which the client requesting solutions of constraints is suspended for the duration of the request. Asynchronous requests are certainly possible due to the inherent parallelism in the specification of constraints and these operations may be parallized.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers - Principles, Techniques and Tools*, Addison Wesley Publishing Company, 1988, pp 203 - 215.

[2] A. Aiba, K. Sakai, Y. Sato, D. Hawley, R. Hasegawa, "Constraint Logic Programming Language CAL", *Procs. FGCS 88*, Tokyo, 1988, pp. 263-276.

[3] Tatsuya Akutsu, Setsuo Ohsuga, "Chemilog - A logic Programming Language/System for Chemical Information Processing", *Procs. FGCS 88*, Tokyo, December 1988, pp. 1176-1183.

[4] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf, "Constraint Hierarchies", *OOPSLA '87 Proceedings*, pp. 48-60.

[5] M.Carlsson, "Freeze, Indexing, and other Implementation Issues in the WAM", *Procs.* 4^{th} *International Conference on Logic Programming*, Volume I, pp. 41-58.

[6] Keith L. Clark, "Logic Programming Schemes", *Procs. FGCS 88*, Tokyo, December 1988, pp. 120-139.

[7] Jacques Cohen, "Constraint Logic Programming Languages", *CACM*, July 1990, pp. 52-68.

[8] Alain Colmerauer, "Prolog in 10 Figures", *CACM*, December 1985, Volume 28, No. 12, pp. 1296-1310.

[9] Alain Colmerauer, "Prolog III", *CACM*, July 1990, pp. 69-90.

[10] Alain Colmerauer, "Opening the Prolog III Universe", *BYTE*, August 1987, pp 177-182.

[11] M. Dincbas, "Constraints, Logic Programming and Deductive Databases", *Programming of Future Generation Computers*, K. Fuchi and M.Nivat (Editors), 1988.

[12] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, F. Berthier, "The Constraint Logic Programming Language CHIP", *Procs. FGCS 88*, Tokyo, December 1988, pp. 693-702.

[13] Francis Giannesini, Henry Kanoui, Robert Pasero, Michel van Caneghem, PROLOG, Addison-Wesley Publishing Company, International Computer Science Series, 1987.

[14] Anthony C. Hearn, "REDUCE USER'S MANUAL - Version 3.2", Rand Publication CP78 (Rev-4/85), April 1985.

[15] N. Heintze, S. Michaylov, P. Stuckey, "CLP(R) and some Electrical Engineering problems", *CMU-CS-89-139*.

[16] N. Heintze, S. Michaylov, P. Stuckey, "CLP(R) and some Electrical Engineering problems", *Procs. ICLP 87*, Melbourne, May 1987, pp. 675-703.

[17] Van Hentenryck, P. and M, Dincbas, "Domains in Logic Programming", *Procs. AAAI-86*, Philadelphia, USA, August 1986, pp. 759-765.

[18] T. J. Hichey, J. Cohen, V. Deschamps, "Meta-Level Interpretation of Constraint Languages – A Case Study: Logical Primitives", *New Generation Computing*, 10 (1992) pp. 361-364.

[19] J. Jaffar, J-L. Lassez, "Constraint Logic Programming", *Procs. POPL 87*, Munich, January 1987, pp. 111-119.

[20] J.Jaffar, J-L. Lassez and M.J. Maher, "A Logic Programming Language Scheme" in: *Logic Programming: Relations, Functions and Equations*, D DeGroot, G. Lindstrom (eds.), Prentice Hall, 1986.

[21] Juhani Jaakola, "Modifying the Simplex Algorithm to a Constraint Solver", *LNCS 456*, pp. 89-105.

[22] Hassan Aït-Kaci, "The WAM: (A Real) Tutorial", *PRL Research Report Number 5*, January 1990.

[23] A. Krall, U. Neumerkel, "The Vienna Abstract Machine", *LNCS 456*, pp. 121-135.

[24] Catherine Lassez, "Constraint Logic Programming", *BYTE*, August 1987, pp. 171-176.

[25] Jiarong Li, "Using Algebraic Constraints in Interactive Text and Graphics Editing", *EUROGRAPHICS '88*, 1988, pp. 197-205.

[26] Wm Leler, "Constraint Programming Languages - Their specification and Generation", 1988.

[27] P. Lim, P. J. Stuckey, "A Constraint Logic Programming Shell", *LNCS 456*, pp. 75-88.

[28] Peret Van Rog, Alvin M Despain, "High Performance Logic Programming with the Aquarius Prolog Compiler", *IEEE Computer*, January 1992.

[29] D. De Schreye, et. al., "Implementing finite-domain constraint logic programming on top of a Prolog System with delay mechanism", *LNCS* 432, 3rd ESOP, pp. 106-117.

[30] Mark Stefik, "Planning with Constraints", (MOLGEN: Part 1), Artificial Intelligence, 16 (1981), pp. 111-140.

[31] Leon Sterling, Ehud Shapiro, "The Art of Prolog - Advanced Programming Techniques", *MIT Press, Cambridge, Massachusetts*, 1986, pp. 206-216.

[32] D.H.D. Warren, "Implementing Prolog - Compiling Predicate Logic Programs, Vol. 1 and 2", *D.A.I. Res. Rep. No. 39 and No. 40*, May 1977.

[33] Stephan Wolfram - "MATHEMATICA - A system for doing mathematics by computer", *Addison Wesley Publishing Company Inc.*, 1988.

Appendix A

Reference Manual

Predicate Name/ Topic — Description

---

abort Aborts the execution of the current goal and returns to the simulator to query the user.

asserta(+Clause)

assertz(+Clause) Both forms of assert add to the predicate database. A clause is added for the predicate in a particular order with respect to existing clauses of that predicate. If the predicate does not exist, it is created by assert. Asserted clauses are not removed by backtracking.

'asserta' adds Clause at the beginning of the predicate definition.

'assertz' adds Clause at the end of the predicate definition.

Refer Example-01, Example-02 and Example-04 in Appendix C.

bye

end_of_file Terminate the current VAM-simulator session.

call(?Goal)

execute(?Goal) Goal must be instantiated to a term that can be interpreted as a goal.

call/1 succeeds if an attempt to prove or satisfy Goal succeeds. The call/1 instance is textually replaced by Goal, i.e. it is transparent. Thus a cut in Goal has an effect at the level of call/1.

execute/1 succeeds if an attempt to prove or satisfy Goal as a subgoal succeeds. A cut in Goal is executed as a subgoal and does not affect choice points at the level of execute/1. Refer to Example-14 in Appendix C.

clause(?Head, ?Body) The expression "clause" refers to a fact or a rule. A clause consists of a head that is positioned at the left of the conditional ":-" and a body that is positioned at the right of same (Head :- Body). The head is made up of a term and the body which is a combination of rule calls. A nonexistent body is equivalent to a body declared as "true".

clause/2 searches the database for a clause whose head matches Head. Head must be bound to a nonvariable term. The head and body of those clauses are unified with Head and Body respectively. If the clause is a unit clause, Body is unified with 'true'. clause/2 backtracks to find all the clauses with matching head.

comment Comments either start with the end of the current line or

/* and extend over arbitrarily many lines until */ is encountered. Nested comments are not allowed.

cut Cut (discard) all choice points created since the parent goal started execution. The cut affects all clauses of the parent goal. Refer to Example-01 and Example-08 in Appendix C.

debug Puts the debugger flag on

nodebug Puts the debugger flag off

fail This predicate always fails. It is useful in the " !, fail " combination. It s also useful when it is desirable to backtrack through all solutons.

help Show help on screen

-Z is +ArithExpr Used for evaluating arithmetic expressions.

listing Lists all clauses in the database.

listing(+Functor) Lists all clauses in the database with the head Functor.

nl Display a newline.

nonvar(+Term) nonvar/1 is true if Term is currently uninstantiated.

numeric(+Term) Arithmetic type testing predicate is true if Term is instantiated to an integer or real value.

retract(Clause) The Head must be sufficiently instantiated to specify a predicate (functor and arity). retract/1 deletes the first clause in the Database that matches clause. As retract is backtrackable, it successively retracts all clauses of the predicate.

stat Shows the current statistics on the console.

struct(+Term) struct/1 is true if Term is a Structure.

term A term is either a structure, a variable or a constant. Integers, reals, atoms, characters, strings, are constants. Terms of the form functor(Arg1, Arg2, ...) and Lists are structures. Every clause in the program is also a term.

true This predicate always succeeds.

unification

?Term1 = ?Term2

+Term1 != +Term2 =/2 is called "unify". If Term1 and Term2 can be unified, they are unified and the predicate succeeds. Do not confuse =/2 with the arithmetic evaluation/assignment operator is/2, and the arithmetic comparison operator ==/2. \= 2 is called "not unifiable". If term1 and Term2 can not be unified, the predicate succeeds. It is defined as the clause "not Term1 = Term2".

univ =..

?Structure =.. - UnivList

-Structure =.. ? UnivList The operator =.. is the univ operator. When this predicate succeeds, UnivList is a list whose head is the atom that is the principal functor of structure and whose tail is the argument list of that functor. At least one of Structure and UnivList must be instantiated when =../2 is invoked. If Structure is uninstantiated, then UnivList must be instantiated either to a list of determinate length (whose head is an atom) or some functor.

Refer to Example-03 in Appendix C. We use the same private symbols which Quintus Prolog provides, an underscore followed by three or more digits. An user cannot refer to particular internal variables by these names, even when using an underscore in the name.

var(+Term) var/1 is true if Term is currently uninstantiated.

variables Variables are distinguished syntactically from other terms by an initial capital letter. A variable is thought of as standing for some particular but unidentified object. A variable is not simply a writable storage location as in most programming languages; but rather it is a local name for some data object.

If a variable is only referred to once, it does not need to be named and may be written as an "anonymous" variable, indicated by the underline "_" alone. This symbol succesfully unifies with anything. Refer to Example-06 in Appendix C. Variables will be internally used either as Prolog Variables or as Constraint Variables.

write/_ Writes out terms on the console. Special write, writeq, display, displayq are not defined.

write_frozen_goals/0 Displays the current set of frozen goals.

write_constraints/0 Displays the current set of unsolved constraints.

```
-----------------------------------------------------------------

   Arithmetic operators : -Result is ..
   ---------------------------------------------------------------
   + +Arith_expr1                  unary plus
   - +Arith_expr1                  unary minus
   +Arith_expr1 + +Arith_expr2     addition
   +Arith_expr1 - +Arith_expr2     subtraction
   +Arith_expr1 * +Arith_expr2     multiplication
   +Arith_expr1 / +Arith_expr2     real division
   +Arith_expr1 // +Arith_expr2    integer division
   +Arith_expr1 mod +Arith_expr2   modulo

    Comparison Predicates
    ------------------------------------------------------------
    +Arith_expr1  <  +Arith_expr2  less than
    +Arith_expr1 =<  +Arith_expr2  less than or equal
    +Arith_expr1 \== +Arith_expr2  Not the same instantiated value
    +Arith_expr1  >  +Arith_expr2  greater than
    +Arith_expr1  >= +Arith_expr2  greater than or equal
    +Arith_expr1  == +Arith_expr2  the same instantiated value
-----------------------------------------------------------------
```

Appendix B

Comparison of VAM with WAM

The Vienna Abstract model is different from other implementation models with respect to :

1. The stack and instruction pointers,
2. The content of stack frames and choice points,
3. The implementation of Unification.

In contrast to VAM, the WAM splits the process of inference into a parameter passing and a unification part. To perform an inference, (a) The parameters are passed via argument registers using put and unify instructions, (b) The control is transferred to the called clause, and (c) The parameters in the argument registers are unified with the arguments of the head using get and unify instructions.
Thus WAM goes :

put, put, ..., call<p>, get, get, ...

VAM makes puts and gets at once. It goes :

c_goal+c_call<p>, g_Any+h_Any, g_Any+h_Any, ...

Whereas WAM creates data superfluously on the copy stack (HEAP) for unifying ground structures which are both in goal and head, VAM creates no terms at all for ground programs. VAM although being a structure copying interpreter, has properties similar to those of structure sharing. The different implementations of inferences influence the memory model, memory utilization, and runtime performance.

WAM's argument registers need to be saved in the choice point and therefore choice point creation and backtracking (especially shallow backtracking) are more expensive when compared to VAM. On backtracking, VAM has to execute put+get instructions of the goal and the next clause. WAM has to execute get instructions only. WAM's overhead of restoring the arguments is approximately equivalent to the "put-overhead" (of fetching g-Any's) in case of VAM. In general, the VAM has fewer trailing and dereferencing operations.

In the VAM, temporary variables cannot be shared between the head and the first subgoal. Variables only occuring in the head and the first goal must be stored as permanent (local) in VAM. Therefore in clauses with more than one subgoal, the stack frame is larger for the call of the first subgoal provided that WAM can share temporaries (typically 2 to 3 elements). In determinate clauses with one subgoal, VAM's increased stack frame is removed by last call optimization. If such a clause is nondeterminate, VAM's stack frame is similar in size to WAM's bigger choice point.

The VAM needs a smaller copy stack size because VAM has no (or fewer) unsafe variables, and because goal structures need not be stored on the copy stack if they are unified with a void variable of the head or with a matching structure.

Appendix C

Test Routines for VAM simulator

C.1 Checking asserta fact

```
/* Example 01 : AAK
   Checking asserta of a fact/clause "factorial(1,1) :- !".
   Executing goal will assert the first clause for
   factorial and enable proper execution of the call
   to factorial later.
*/
factorial(N,Res) :-
   N1 is N-1, factorial(N1,Temp), is(Res,*(Temp,N)).
goal :- asserta((factorial(1,1) :- !)),fail.
goal :-
   listing(factorial),
   factorial(5,X),
   write(X), nl.

Goal : goal.
-----------------------SIMULATING----------------------
factorial(1, 1) :-
   !.
factorial(N, Res) :-
   is(N1, -(N, 1)),
   factorial(N1, Temp),
   is(Res, *(Temp, N)).
120
--------------- xxx ---------------- xxx --------------
Frozen Goals:□                              Constraints :□
```

C.2 Checking assertz fact

```
/* Example 02 : AAK */
/* Checking asserta and assertz and cuts */

% First execute goal1 to check asserta,
% then execute goal2 to execute assertz

goal1 :- assertz((work(A,bbb(ccc)) :- true)),
         work('c,bbb(_)).
goal2 :- asserta(
    (work(A,bbb(ccc)) :-
       work1(A),work2(B),fail
    )
                ),
         work('c,bbb(_)).
work1(X) :- write(X,"Work1"),nl,!.
work1(X) :- write("Should never come here"),nl.
work2(X) :- write(X,"Work2"),nl.
work2(X) :- write("Should always come here"),nl.

work(C,D) :- write(C,D,"End of execution").

Goal : goal1.
-----------------------SIMULATING----------------------
aaa
Warning : AAK : Asserting a clause containing variables
Created temporary variables : [TV0]
'cbbb(_399)"End of execution"
--------------  xxx  -------------- xxx  -------------
Frozen Goals:[]                          Constraints :[]

Goal : goal2.
-----------------------SIMULATING----------------------
aaa
Warning : AAK : Asserting a clause containing variables
Created temporary variables : [TV0]
'c"Work1"
_590"Work2"
"Should always come here"
'cbbb(_267)"End of execution"
--------------  xxx  -------------- xxx  -------------
Frozen Goals:[]                          Constraints :[]
```

C.3 Towers of Hanoi using a memo-function

Memo functions save the results of subcomputations to be used later in a computation. Remembering partial results is impossible within Pure Prolog, so memo-functions are implemented using side effects provided by assert to the program. Programming in this way may be considered bottom up programming.

The prototypical memo-function is *lemma(Goal)*. Operationally it attempts to prove the goal *Goal*, and if successful, stores the result of the proof as a lemma (See clause for *lemma* below demonstrated for solving the Towers of Hanoi problem). The next time P is attempted, the new solution will be used, and there will be no unnecessary computation. The cut is present to prevent the more general program being used. Its use is justified only if P does not have multiple solutions.

```
hanoi(1,A,B,C,[A to B]).
hanoi(N,A,B,C,Moves) :- N > 1,
   N1 is N-1,
   lemma(hanoi(N1,A,C,B,Ms1)),
   hanoi(N1,C,B,A,Ms2),
   append(Ms1,[A to B|Ms2],Moves).
append([],Z,Z) :- !.
append([X|Y],Z,[X|Rest]) :- append(Y,Z,Rest).
lemma(P) :- call(P),asserta((P :- !)).
testhanoi(N,Pegs,Moves) :-
   hanoi(N,A,B,C,Moves), Pegs=[A,B,C].
```

It is well known that the solution of the Towers of Hanoi with N disks requires 2^{N-1} moves. The performance is dramatically improved in the above program.

The solution to the Towers of Hanoi repeatedly solves subproblems moving the identical number of disks. A memo-function can be used to recall the moves made in solving each subproblem of moving a smaller number of disks. Later attempts to solve the subproblem can use the computed sequence of moves rather than recomputing them.

The program is tested with the *testhanoi(N,Pegs,Moves)*. *N* is the number of disks, *Pegs* is a list of three peg names and *Moves* is the list of moves that must be made. Note that in order to take advantage of the memo-functions, a general problem is solved first. Only when the solution is complete, and all memo-functions have recorded their results, are the peg names instantiated.

```
Goal :testhanoi(4,[a,b,c],Moves).
-----------------------SIMULATING----------------------
Warning : AAK : Asserting a clause containing variables
Created temporary variables : [TV0,TV1,TV2]

Warning : AAK : Asserting a clause containing variables
Created temporary variables : [TV0,TV1,TV2]
```

```
Warning : AAK : Asserting a clause containing variables
Created temporary variables : [TV0,TV1,TV2]
--------------- xxx --------------- xxx ---------------
Frozen Goals:[]                                Constraints :[]
Moves : [to(a, c), to(a, b), to(c, b), to(a, c), to(b, a),
         to(b, c), to(a, c), to(a, b), to(c, b), to(c, a),
         to(b, a), to(c, b), to(a, c), to(a, b), to(c, b)]
```

In the process the following three clauses have been added. (This is found by giving the goal *listing(hanoi)*).

```
hanoi(3, TV0, TV1, TV2,
   [to(TV0, TV1), to(TV0, TV2), to(TV1, TV2),
    to(TV0, TV1), to(TV2, TV0), to(TV2, TV1),
    to(TV0, TV1)]) :- !.
hanoi(2, TV0, TV1, TV2,
   [to(TV0, TV2), to(TV0, TV1), to(TV2, TV1)]) :- !.
hanoi(1, TV0, TV1, TV2, [to(TV0, TV1)]) :- !.
```

C.4 Using structures and unnamed variables

```
/* Example 03 : AAK */
/* Checking multiple "_"es and use of structures */

goal :- person(Name,_,_,_,dateofbirth(_,month(6),_)),
        write(Name),nl,fail.
goal.

person("Jonny",m,age(23),hobbies([reading,swimming]),
       dateofbirth(day(9),month(6),year(1968))).
person("Robert",m,age(15),hobbies([reading,swimming]),
       dateofbirth(day(19),month(7),year(1985))
      ).
person("Meena",f,age(20),hobbies([eating,swimming]),
       dateofbirth(day(15),month(6),year(1975))
      ).

Goal : goal.
-----------------------SIMULATING----------------------
"Jonny"
"Meena"
--------------- xxx --------------- xxx ---------------
Frozen Goals:[]                                Constraints :[]
```

C.5 Checking assertion of a clause with variables

```
/* Example 4 : AAK */
/* Checking asserta of clauses with variables */
new_hi(X) :- write(X),nl.
goal :-
   asserta((
      hi(X):-
         write("Executing the code ",
               '(new_hi(X) :- write(X),nl)',
% Note: What happens if the single quotes making the above line
% an atom are missing?
% Answer : These X also get bounded when hi(123) is called.
               "with X = 123."
              ),
         nl, new_hi(X)
          )),
   hi(123), write("Done"),nl.

Goal : goal.
-----------------------SIMULATING----------------------
aaa
Warning : AAK : Asserting a clause containing variables
Created temporary variables : [TV0]
"Executing the code "(new_hi(X) :- write(X),nl)"with X = 123."
123
"Done"
---------------  xxx  --------------- xxx  --------------
Frozen Goals:□                                Constraints :□
```

C.6 Checking Univ =..

```
/* Example 06 : AAK */
/* Checking =.. Univ */
goal :-
   aaa(X,ccc) =.. [F,bbb,ccc], write(X),nl, write(F).
Goal : goal.
-----------------------SIMULATING----------------------
bbb
aaa
---------------  xxx  --------------- xxx  --------------
Frozen Goals:□                                Constraints :□
```

C.7 Checking call and execute

```
/* Example 14 : AAK
   Testing call and execute.
*/

goal1 :- X = (!,fail), write(test1),call(X).
goal1 :- write(test2).

goal2 :- X = (!,fail), write(test1),execute(X).
goal2 :- write(test2).

Goal : goal1.
-----------------------SIMULATING----------------------
test1
[Result:Unsuccessful.]

Goal : goal2.
-----------------------SIMULATING----------------------
test1test2
--------------- xxx --------------- xxx -------------
Frozen Goals:□                          Constraints :□
```

C.8 Sorting : Checking cuts

```
/* Using Cuts : Example 08 : AAK
   Making programs deterministic */
   Problem : Sort a given list of integers.
*/

strictmaxoftwo(X,Y,Y) :- Y > X,true.
strictmaxoftwo(X,Y,X) :- >(X,Y).

ismaxinlist(□,□).
ismaxinlist([Num],Num) :- !.
ismaxinlist([Num|Rem],Max) :-
   Rem ismaxinlist TempMax,
   strictmaxoftwo(Num,TempMax,Max).

removefromlist(□,_,□).
removefromlist([Max|Rem],Max,Rem) :- !.
removefromlist([Num|Rem],Max,[Num|NewRem]) :-
   removefromlist(Rem,Max,NewRem).
```

```
sort([],[]) :- !.
sort(X,[Max|Rem]) :-
   ismaxinlist(X,Max),
   removefromlist(X,Max,TempRem),
   sort(TempRem,Rem),!.

goal :- goal1, goal2, write('Goal1 and Goal2 executed'), !, fail.
goal1 :- sort([3,2,4,1],X),write(X),nl,fail.
goal2 :- sort([10,12,11],[12,X,Y]),sort([X,4,Y],Z),write(Z),nl,!.
goal1 :- sort([],Y),sort([5,6,8,7],X),write(X),nl,!.
goal2 :- write("Should never reach here"),fail.
goal :- write("Second Goal"),nl,!.

Goal : goal.
-----------------------SIMULATING----------------------
[4, 3, 2, 1]
[8, 7, 6, 5]
[11, 10, 4]
Goal1 and Goal2 executed
[Result:Unsuccessful.]
```

C.9 Permutations : Using dif and freeze

```
/* Example 13 : AAK */
/* Shows use of freeze and dif predicates */
plus(A,B,C) :-
   freeze(A, freeze(B, either(U, sum1(A,B,C)))),
   freeze(A, freeze(C, either(U, sum2(A,B,C)))),
   freeze(B, freeze(C, either(U, sum3(A,B,C)))).
sum1(A,B,C) :- C is A + B.
sum2(A,B,C) :- B is C-A.
sum3(A,B,C) :- A is C-B.
either(U,P) :- var(U),!,U=constant,call(P).
either(_,_).

goal1 :-
   plus(3,5,Z),write(Z),nl,
   plus(3,Y,8),write(Y),nl,
   plus(X,5,8),write(X),nl.

list_of_one([1|X]) :- write('.'),list_of_one(X).
list_of_one([]).
length(L,N) :- freeze(L,length_new(L,N)).
```

```
length_new([],0).
length_new([E|L],N) :-
   dif(N,0), one_less(N,N_new), length(L, N_new).
one_less(X,Y) :- Y is X - 1.
dif(X,Y) :- freeze(X, freeze(Y, X \= Y)).

goal2 :- length(L,5), list_of_one(L), write(L),nl.

outside_of(X,[]).
outside_of(X,[Y|L]) :- dif(X,Y), outside_of(X,L).
not_same([X|L]) :- outside_of(X,L), not_same(L).
not_same([]).
permutation(L,N) :- length(L,N), not_same(L), all_digits(L).
all_digits([]).
all_digits([X|L]) :- dig(X), all_digits(L).
dig(1).
dig(2).
dig(3).
dig(4).
dig(5).

goal3 :- asserta(count(0)), fail.
goal3 :-
   permutation(L,4),write(L),nl,
   retract(count(N)),
   NewN is N + 1,
   asserta(count(NewN)),
   fail.
goal3 :- retract(count(N)),write('Count = '),write(N),nl.

Goal : goal1.
-----------------------SIMULATING----------------------
8
5
3
--------------  xxx  -------------- xxx  --------------
Frozen Goals:[]                        Constraints :[]

Goal : goal2.
-----------------------SIMULATING----------------------
.....[1, 1, 1, 1, 1]
--------------  xxx  -------------- xxx  --------------
Frozen Goals:[]                        Constraints :[]

Goal : goal3.
```

```
------------------------SIMULATING----------------------
[1, 2, 3, 4]
[1, 2, 3, 5]
[1, 2, 4, 3]
   . . .
[5, 4, 3, 1]
[5, 4, 3, 2]
Count = 120
---------------  xxx  --------------- xxx  --------------
Frozen Goals:[]                          Constraints :[]
```

Appendix D

Test Programs which use Constraint Solvers

D.1 Testing real black-box solvers using RPC

This program demonstrates that domain real solvers can be executing on any/or all of the machines, say in our case tejas and vayu. Also more than one real solver can be executing on the same machine with different program numbers for RPC's. Any of these can be invoked by the user, simplify by specifying the name of the machine and the program number in the *goal*.

```
/* Example 101  : AAK */
/* Checking of REAL black_box constraint solver on tejas or vayu
   for different constraint_prog's.
*/
goal :-
   write('Please specify goal(MachineName,ConstraintProg).'),
   write('eg: goal(tejas,99).'),nl,
   write('    goal(tejas,98).'),nl,
   write('    goal(vayu,99).'),nl.

goal(MachineName,N) :-
   {[[real,rpc,machine_name=MachineName,constraint_prog=N],
     [RealA+RealB=2.0,RealA-RealB+RealC=1.0]
   ]},
   RealC = 1.0,
   write('Solved  '),write('RealA='),write(RealA),
                     write(', RealB='),write(RealB),
                     write(', RealC='),write(RealC),nl.
```

The output of this program on any of the solvers is :

```
Solved  RealA=1.0, RealB=1.0, RealC=1.0
```

D.2 Using float and real solvers to demonstrate structure independence

This program is used to show that the domain of the constraints specified could be any domain. Thus the constraints are written independent of the domain thus showing structure independence. We have used float solver and the real solvers which are different domains. Real solvers require that the constants be atoms while float require them to be numeric. However the filter for the real solver takes care of converting the numeric constants to atoms on the solver side. Therefore the program works for either of the domains.

```
/* Example 102 : AAK */
/* Checking of float, real solvers on tejas or vayu for any details
   specified directly by the user.Notice that the domain i.e. float
   or real may also be specified online by the user thus making the
   specifications structure independent.
   Also notice that unification is used for RealC  with  constraint
   solving for finding solutions for the three variables.
*/
goal :-
   write('Please specify goal(DetailsOfSolver).'),nl,
   write('Examples:'),nl,
   write('    goal(float). % Default internal solver.'),nl,
   write('    goal(real).  % Default remote tejas.'),nl,
   write('    goal([real,rpc,machine_name=vayu]).'),nl,
   write('The complete list of details that could be'),nl,
   write('specified when using rpcs is :'),nl,
   write('   machine_name, constraint_prog, constraint_vers,'),
   write('   function_number, tcp_udp.'),nl,
   write('NOTE: This  may  also be  used for incremental'),nl,
   write('      solvers but fail must be given after the'),nl,
   write('      goal to allow the incremental constraint'),nl,
   write('      solver to go back to its original state.'),nl.
goal(ConstraintDetail) :-
   {[ConstraintDetail,[RealA+RealB=2.0,RealA-RealB+RealC=1.0]
   ]}, RealC = 1.0,
   write('Solved  '),write('RealA='),write(RealA),
                     write(', RealB='),write(RealB),
                     write(', RealC='),write(RealC),nl.
```

The output of the program is :

```
Solved  RealA=1.0, RealB=1.0, RealC=1.0
```

D.3 Checking boolean black-box solver

The boolean black-box solver is invoked through same_fs strategy. The different goals illustrated below show normal solutions as well as backtracking.

```
/* Example 103 : AAK */
/* Checking of boolean constraint solver */
% Checking true solution
goal1 :-
   write(X),write('   '),write(Y),nl,
   {[boolean,[(true and false) = X, true = (X or Y)]]},
   write('The values of X and Y are :'),
   write(X),write('   '),write(Y),nl.

% Checking whether variables are printed properly
goal2 :-
   write(X),write('   '),write(Y),write('   '),write(Z),nl,
   {[boolean,[(true and false) = X, true = (X or Y or Z)]]},
   write('The values of X, Y and Z are :'),
   write(X),write('   '),
   write(Y),write('   '),
   write(Z),nl.

% Checking execution with fail
goal3 :-
   write('Checking whether true = false'),nl,
   {[boolean,[true = false]]},
   write('Checked that true = false.'),nl.
goal3 :-
   write('True != False.'),nl.
```

The outputs are as follows :

```
goal1 :
   The values of X and Y are :false   true

goal2 :
   _X   _Y   _Z
   The values of X, Y and Z are :false   _Y   _Z
   Constraints : [[boolean, [=(true, or(_Y, _Z))]]]

goal3 :
   True != False.
```

D.4 Checking real incremental solver

The incremental solver for domain real is simulated on the nonincremental solver by stacking constraints when a new constraint is added and unstacking for backtracking. Though slow, it suffices to illustrate the working and interfacing of incremental solvers. Different goals are illustrated below :

```
/* Example 104 : AAK */
/* CHECKING OF THE REAL INCREMENTAL SOLVER ON TEJAS */

goal :- write('Try goal10, goal11, goal12, and goal13.'),nl.
%-------------------------------------------------------------
% Backtracking of incremental solver because of later fail.
goal10:-
   {[inc_real,[A+B=9.0]]},
   write('A after first step :'),write(A),nl,
   write('B after first step :'),write(B),nl,
   {[inc_real,[A-B=1.0]]},
   write('Final values are :'),write(A),nl,write(B),nl,fail.
goal10.

%-------------------------------------------------------------
% Backtracking of incremental solver because of inconsistency
% and then because of fail.
goal11:-
   goaltry1(A,B),
   write('A after first step :'),write(A),nl,
   write('B after first step :'),write(B),nl,
   {[inc_real,[A-B=0.0]]},
   write('Final values are :'),write(A),write('  '),
                              write(B),nl,
   {[inc_real,[A+1.0=B]]}.
            /* Actually A = B, This causes fail. */
goal11 :- true.
goaltry1(A,B) :-
   {[inc_real,[A+B=10.0]]}.
goaltry1(A,B) :-
   {[inc_real,[A+B=8.0]]}.

%-------------------------------------------------------------
% Backtracking in solver in presence of cut-fail combination.
goal12 :- goal12a.
goal12 :- goal12b,fail.
goal12.
```

```
goal12a:-
   {[inc_real,[A+B=10.0]]},
   {[inc_real,[A-B=0.0]]},!,
   write('Final values are :'),write(A),write('  '),
                              write(B),nl,fail.
goal12b:-
   {[inc_real,[A+B=10.0]]},
   {[inc_real,[A-B=2.0]]},
   write('Final values are :'),write(A),write('  '),
                              write(B),nl.

%-------------------------------------------------------------
/* The final fail in goal1 is to reset the state of the
   constraint solver to where it was when the program started.
   Notice that although there is a cut at the end of clause 2
   of work/3, the backtrack call is still given to the
   constraint solver because of fail in goal1. Thus cuts have
   no effect on the correct sending of backtracking calls to
   the incremental solver thus allowing the solver to maintain
   its state correctly, even in presence of cuts. Thus correct
   backtrack calls which were supposed to be given to the
   incremental solver without the presence of cuts will also
   be given in the presence of cuts.
*/
goal13 :- goal(bbb), fail.
    % We may have a cut before the fail here.
    % This also has no effect on backtrack calls to solver.
% Notice no clause to make goal3 true, so Unsuccessful,
% but solver state is proper.
goal(U) :- work(U,A,B),
   write('A='),write(A),nl,write('B='),write(B),nl.

work(AAA, A, B) :-
   {[inc_real,[A+B=10.0, A-B=2.0]]}, AAA = aaa.
work(BBB, A, B) :-
   {[inc_real,[A+B=10.0, A-B=6.0]]}, BBB = bbb,!.
% This will never be executed because of the cut in clause above.
work(BBB, A, B) :-
   {[inc_real,[A+B=10.0, A-B=8.0]]}, BBB = bbb.
%-------------------------------------------------------------
```

The outputs for each of the goals is given below along with statements indicating what is happening internally.

goal11:

```
   Solver called : Backtrack must be sent.
A after first step :
   Solver called :
      Output command, so no backtrack must be sent for this.
_A
B after first step :
   Solver called :
      Output command, so no backtrack must be sent for this.
+(9.0, *(-1.0, _A))
   Solver called : Backtrack must be sent.
Final values are :
   Solver called :
   Output command, so no backtrack must be sent for this.
5.0
   Solver called :
   Output command, so no backtrack must be sent for this.
4.0
Backtrack call given to incremental solver
Backtrack call given to incremental solver
```

The solutions of the other goals are given along with positions of backtrack calls.

```
goal12:
   Final values are : 5.0 5.0
      Backtrack call given to incremental solver
      Backtrack call given to incremental solver
   Final values are : 6.0 4.0
      Backtrack call given to incremental solver
      Backtrack call given to incremental solver

goal13:
      Backtrack call given to incremental solver
   A= 8.0
   B= 2.0
      Backtrack call given to incremental solver
```

D.5 Checking multiple solvers in action

Quintus Prolog is invoked two times for this problem. The inference engine along with its float solver is executing on csesun1 on the first invocation of Quintus Prolog. Domain boolean constraint solver is executing on csesun1 on the second invocation of Quintus Prolog. Two similar Domain real constraints solvers are executing on tejas and vayu.

Notice that :

1. The constraint variable RealC has been used across the two real constraint solvers, i.e. on tejas and vayu. RealC is passed to tejas as well as unified by the inference engine.

2. The domain boolean solver by default takes the same_fs (same file system) method for communication through files. (Current default file number is 10).

3. The domain real by default takes the tejas solver with rpc used for communication. The constraint_prog is 99, constraint_vers is 1 and function_number is 1. 'tcp' is used for rpc's.

4. Initially the boolean solver is called with BoolVal uninstantiated and later when BoolVal is unified with false, then the constraint gets solved by the boolean constraint solver. Thus there are multiple calls to the boolean constraint solver.

5. The float solver is attached to the inference engine and it is called more than once since the constraints are specified separately.

```
goal :-
   Details = [real,rpc,machine_name=vayu,
                       constraint_prog=99,
                       constraint_vers=1,
                       function_number=1,
                       tcp_udp=tcp
             ],
   {[boolean,[true = '~'(BoolVal)]],
    [Details,[FRealA+FRealB+FRealC='4.0',
              FRealA-FRealB='0.0',
              FRealC=RealC-'1.0']],
    % RealC gets bound to '1.0' because of above solver
    [real,[RealA+RealB='2.0',RealA-RealB+RealC='1.0']],
    [float,[FloatA+FloatB+FloatC=6.0,FloatA-FloatB= -1.0]]
   },
   BoolVal = false,RealC = '1.0',
   {[float,[FloatC-FloatB = 1.0]]},
  write('Solved         :'),nl,
  write('Boolean        : '),
      write(true = '~'(BoolVal)),nl,
  write('RealVayu       : '),
      write('FRealA='),write(FRealA),
      write(', FRealB='),write(FRealB),
      write(', FRealC='),write(FRealC),nl,
  write('RealTejas      : '),
      write('RealA='),write(RealA),
      write(', RealB='),write(RealB),
```

```
    write(', RealC='),write(RealC),nl,
write('FloatInternal : '),
    write('FloatA='),write(FloatA),
    write(', FloatB='),write(FloatB),
    write(', FloatC='),write(FloatC),nl.
```

The final output is as follows :

```
Boolean       : =(true, ~(false))
RealVayu      : FRealA=2.0, FRealB=2.0, FRealC=0.0
RealTejas     : RealA=1.0, RealB=1.0, RealC=1.0
FloatInternal : FloatA=1.0, FloatB=2.0, FloatC=3.0
```

Appendix E

VAM-Code

The code is produced in a form that the simulator written in Prolog can understand. Thus the whole code is read as a single term. We have used internal codes as follows : v(1) for fstvar, v(2) for nxtvar, v(3) for constant, v(4) for struct and v(5) for list. We use temporaryvar for storing names but these are converted to internal variables before use. Also note that in the current implementation, we have used v(3), const(nill) instead for the instruction nil.

E.1 Code for Installments Capital Problem

The VAM code produced by the compiler written using C (LEX and YACC) for the installments capital problem described in Figure 4.1 is shown here.

```
[%Procedure:001,Clause:001==================================
vam_clause([v(3), const(atom(('goal'))), c_goal,
v(4), functor(('installments_capital'),2),
v(5), v(1), temporaryvar('I'),
v(5), v(4), functor(('*'),2),
v(3), const(real(2.000000)),
v(2), temporaryvar('I'),
v(5), v(4), functor(('*'),2),
v(3), const(real(3.000000)),
v(2), temporaryvar('I'),
v(3), const('nill'),
v(3), const(real(1000.000000)),
c_call, c_goal,
v(4), functor(('write'),1),
v(3), const(str('Answer:   I=')),
c_call, c_goal,
v(4), functor(('write'),1),
v(2), temporaryvar('I'),
c_call, c_goal,
```

```
v(3), const(atom(('nl'))),
c_lastcall],['I']),
%Procedure:002,Clause:001==================================
vam_clause([v(4), functor(('installments_capital'),2),
v(5), v(1), temporaryvar('I'),
v(1), temporaryvar('X'),
v(1), temporaryvar('C'), c_goal,
v(4), functor(('merge_constraints'),1),
v(4), functor(('='),2),
v(1), temporaryvar('Y'),
v(4), functor(('-'),2),
v(4), functor(('*'),2),
v(3), const(real(1.100000)),
v(2), temporaryvar('C'),
v(2), temporaryvar('I'),
c_call, c_goal,
v(4), functor(('installments_capital'),2),
v(2), temporaryvar('X'),
v(2), temporaryvar('Y'),
c_lastcall],['Y', 'C', 'X', 'I']),
%------------Clause:002-----------------------------------
vam_clause([v(4), functor(('installments_capital'),2),
v(3), const('nill'),
v(3), const(real(0.000000)),
c_nogoal],[])].
```

E.2 Code for Permutations

A part of the VAM code produced by the compiler for Example 13 in Appendix C is included here.

```
            . . .
%Procedure:011,Clause:001==================================
vam_clause([v(4), functor(('length'),2),
v(1), temporaryvar('L'),
v(1), temporaryvar('N'), c_goal,
v(4), functor(('freeze'),2),
v(2), temporaryvar('L'),
v(4), functor(('length_new'),2),
v(2), temporaryvar('L'),
v(2), temporaryvar('N'),
c_lastcall],['N', 'L']),
            . . .
```

Appendix F

Real World Programs using Solvers

F.1 Factorial

The program for finding factorial is shown below :

```
factorial(A,B) :- {A=0.0,B=1.0}.
factorial(A,B) :- {B=A*FactAminus1, Aminus1=A-1.0},
                  factorial(Aminus1,FactAminus1).
```

This simple factorial procedure uses constraints to :

1. Find the factorial B of a given number A. e.g. goal1,goal4
2. Find the number A whose factorial B is given. e.g. goal2,goal5
3. Given A and B, Check if number B is factorial of A. e.g. goal3
4. Generate A and B both if both are variables e.g. goal0

Each of the goals numbered 1 to 5 are shown below :

```
goal0 :- factorial(X,Y),
   write('Solution of factorial(X,Y) is X='),write(X),
   write(', Y='),write(Y),nl,fail.
goal1 :- factorial(3.0,X),
   write('Solution of factorial(3.0,X) is X='),write(X),nl.
goal2 :- factorial(X,6.0),
   write('Solution of factorial(X,6.0) is X='),write(X),nl.
goal3 :- factorial(3.0,6.0),
   write('Solution of factorial(3.0,6.0) is '),nl.
goal4 :- factorial(4.0,X),
   write('Solution of factorial(4.0,X) is X='),write(X),nl.
goal5 :- factorial(X,24.0),
   write('Solution of factorial(X,24.0) is X='),write(X),nl.
```

Solutions for each of the above goals are as follows :

```
goal0:
   Solution of factorial(X,Y) is X=0.0, Y=1.0
   Solution of factorial(X,Y) is X=1.0, Y=1.0
   Solution of factorial(X,Y) is X=2.0, Y=2.0
   Solution of factorial(X,Y) is X=3.0, Y=6.0
   Solution of factorial(X,Y) is X=4.0, Y=24.0
   Solution of factorial(X,Y) is X=5.0, Y=120.0
                . . .
goal1: Time taken: 0.767 sec
   Solution of factorial(3.0,X) is X=6.0
goal2: Time taken: 3.683 sec
   Solution of factorial(X,6.0) is X=3.0
goal3: Time taken: 0.667 sec
   Solution of factorial(3.0,6.0) is
goal4: Time taken: 0.967 sec
   Solution of factorial(4.0,X) is X=24.0
goal5: Time taken: 8.234 sec
   Solution of factorial(X,24.0) is X=4.0
```

The factorial program has been extended below to handle even wrong factorials.

```
fact_solve(FactOf,Value) :-
   numeric(FactOf),!, factorial(FactOf,Value).
fact_solve(FactOf,Value) :-
   numeric(Value),!, loop(1.0,Value,FactOf).
loop(X,Value,FactOf) :-
   factorial(X,Y), check_fact(X,Y,Value,FactOf).
check_fact(X,Y,Value,X) :- {Y=Value},!.
check_fact(X,Y,Value,FactOf) :-
   {Y<Value},!,NewX is X+1.0,loop(NewX,Value,FactOf).
check_fact(X,Y,Value,notPossible) :- {Y>Value},!.
check_fact(X,Y,Y,FactOf) :-
    numeric(X),numeric(FactOf),X==FactOf,!.
check_fact(X,Y,Value,FactOf) :-
   numeric(X),numeric(FactOf),X<FactOf,!,
   NewX is X+1.0,loop(NewX,Value,FactOf).
```

The goals numbered 6 to 9 are used to test this as shown below :

```
goal6 :- fact_solve(X,6.0),
   write('fact_solve(X,6.0) gives X='),write(X),nl.
goal7 :- fact_solve(3.0,X),
   write('fact_solve(3.0,X) gives X='),write(X),nl.
goal8 :- fact_solve(3.0,6.0),
```

```
   write('fact_solve(3.0,6.0) gives'),nl.
goal9 :- fact_solve(X,7.0),
   write('fact_solve(X,7.0) gives X='),write(X),nl.
```

The solutions for the goals numbered 6 to 9 are given below :

```
goal6: Time taken: 3.266 sec
   fact_solve(X,6.0) gives X=3.0
goal7: Time taken: 0.850 sec
   fact_solve(3.0,X) gives X=6.0
goal8: Time taken: 0.717 sec
   fact_solve(3.0,6.0) gives
goal9: Time taken: 5.000 sec
   fact_solve(X,7.0) gives X=notPossible
```

F.2 Complex numbers

Shown below is a program for multiplying two complex numbers.

```
/* Multiply complex numbers */
c_mult(c(R1,I1),c(R2,I2),c(R3,I3)):-
   {R3 = R1 * R2 - I1 * I2, I3 = R1 * I2 + R2 * I1}.
goal:-
   c_mult(c(1.0,1.0),c(2.0,2.0),Z),
   c_mult(c(1.0,1.0),Y,c(0.0,4.0)),
   c_mult(X,c(2.0,2.0),c(0.0,4.0)),
   write(X),nl,write(Y),nl,write(Z),nl.
```

The solution for the above goal is :

```
c(1.0, 1.0)
c(2.0, 2.0)
c(-0.0, 4.0)
Time taken : 0.150 sec
```

F.3 Light Meals Problem - Let's Eat Well

Consider a problem to devise a possible set of light meals. Let H be the appetizer (horsDceuvre), M be the main course and D be the dessert. Then the triplet *H,M,D* is a meal. The light meal is defined by assigning a certain number of caloric units to each course and restricting itself to meals that add up to a number of units smaller than 10. The listing of the program for devising light meals is shown below :

```
/* Computing Light meals */
   light_meals(H,M,D):-
      {[real,[I>=0.0,J>=0.0,K>=0.0,I+J+K=<10.0]]},
```

```
    horsDceuvre(H,I),
    mainCourse(M,J),
    dessert(D,K).

mainCourse(M,I):-meat(M,I).
mainCourse(M,I):-fish(M,I).

horsDceuvre(radishes,1.0).
horsDceuvre(pate,6.0).

meat(beef,5.0).
meat(pork,7.0).

fish(sole,2.0).
fish(tuna,4.0).

dessert(fruit,2.0).
dessert(icecream,6.0).

goal :-
    light_meals(H,M,D),
    write('H='),write(H),
    write(', M='),write(M),
    write(', D='),write(D),nl,
    fail. /* For listing all solutions */
```

The query light_meals(H,M,D) in the goal above allows us to obtain all the sets of H, M and D that constitute a light meal. In this case there are six replies. The solutions are listed below :

```
H=radishes, M=beef, D=fruit
H=radishes, M=pork, D=fruit
H=radishes, M=sole, D=fruit
H=radishes, M=sole, D=icecream
H=radishes, M=tuna, D=fruit
H=pate, M=sole, D=fruit
```

Time taken for the above problem on the internal float solver is 4.116 sec. In the predicate light_meals above, the constraints are set up at the very beginning. Thus, the same constraints will be checked again and again, initially with variables unbound and later when the variables get bound. This seems to introduce a lot of overhead over the normal method we would use to write the same program in Prolog in which the constraints would be checked at the end of the predicate light_meals after all variables have been bound as shown below :

```
light_meals(H,M,D):-
```

```
      horsDceuvre(H,I),
      mainCourse(M,J),
      dessert(D,K),
      I>=0.0,J>=0.0,K>=0.0,I+J+K=<10.0 .
% or  {[real,[I>=0.0,J>=0.0,K>=0.0,I+J+K=<10.0]]}.
```

This is the generate-and-test technique. Time taken on the internal float solver is 3.417 sec. The inefficiency is present in this small example which is used only to demonstrate the use of constraints. However, for larger problems this forward checking with the help of constraints increases the efficiency of the program since incorrect values are never selected. This is illustrated in the next example.

F.4 Cryptarithmetic puzzle

Consider the classical cryptarithmetic addition program. For given strings of letters "SEND", "MORE" and "MONEY", each of which represent different integers from 0 to 9, the problem is to find an appropriate assignment of digits for each letter so that adding the numbers represented by "SEND" and "MORE" yields the number represented by "MONEY". Consider a Prolog program using the generate and test strategy for solving the above puzzle.

```
solution([S,E,N,D],[M,O,R,E],[M,O,N,E,Y]) :-
   admissible(R1,0,0,M,0), admissible(R2,S,M,O,R1),
   admissible(R3,E,O,N,R2), admissible(R4,N,R,E,R3),
   admissible(O,D,E,Y,R4),
   without_repetition([S,E,N,D,M,O,R,Y]),
   M \== 0, S \== 0.
admissible(R,X,Y,Z,R1) :-
   remainder(R), digit(X), digit(Y),
   T is R+X+Y, R1 is T/10, Z is (T mod 10).
remainder(0).
remainder(1).
digit(0).
 ...
digit(9).
without_repetition([]).
without_repetition([X|Y]) :- out_of(X,Y),
   without_repetition(Y).
out_of(X,[]).
out_of(X,[A|L]) :- X \== A, out_of(X,L).
```

This program is inefficient under Prolog's standard left to right control rule. The program is rewritten using forward checking with the help of constraints as shown below :

the process. This shows that the use of coroutining helps in finding the solutions faster. The solution to this problem may also be rewritten using the predicate *dif* instead of constraints by using the predicate *freeze*, because all that is required is that is of the values assigned to S, E, N, D, M, O, R, Y are different. The the definition of out_of will be :

```
out_of(_,[]) :- !.
out_of(U,[V|W]) :-
   out_of(U,W), dif(U, V).
dif(X,Y) :- freeze(X, freeze(Y, X \== Y)).
```

Also instead of `S \== 0` and `M \== 0` in clause for *solution* we use dif(S,0) and dif(M,0) respectively. The same solutions are obtained.

F.5 Computing Scalar Products

The following program is used to compute scalar products.

```
/* Computing Scalar Products */
/* Refer Pg 79, CACM July 90 */
scalar_product([],[],Value) :- {Value=0.0}.
scalar_product([X1|X],[Y1|Y],W) :-
   {W1=X1*Y1,W=W1+Z},
   scalar_product(X,Y,Z).
goal :-
   scalar_product([1.0,1.0],X,12.0),
   scalar_product(X,[2.0,4.0],34.0),
   write('Answer is X = '),write(X),nl.
```

The solution of goal above is :

```
Answer is X = [7.0, 5.0]
Time taken: 1.033 sec
```

F.6 Trajectory Problem

The problem is to find the points on the trajectory.

```
traj([_,_]).
traj([Hm, H, Hp|T]):-
   merge_constraints(,(Hp-H=H-Hm-G,G=1.0)),
   traj([H,Hp|T]).
goal :-
      X1 is 0.0, X10 is 5.0,
      traj([X1,X2,X3,X4,X5,X6,X7,X8,X9,X10]),
      write('X1='),write(X1),nl,
```

```
        write('X2='),write(X2),nl,
        write('X3='),write(X3),nl,
        write('X4='),write(X4),nl,
        write('X5='),write(X5),nl,
        write('X6='),write(X6),nl,
        write('X7='),write(X7),nl,
        write('X8='),write(X8),nl,
        write('X9='),write(X9),nl,
        write('X10='),write(X10),nl.
```

Solutions are :

```
    X1=0.0
    X2=4.55555
    X3=8.1111
    X4=10.6666
    X5=12.2222
    X6=12.7777
    X7=12.3333
    X8=10.8888
    X9=8.4444
    X10=5.0
Time taken: 4.766 sec
```

F.7 Circuit Solver

Note : This file will be loaded by dc_circ1 and dc_circ2 independently. The constraints will be solved by the float solver attached to the inference engine on the client itself. The constraints will not be passed to any remote machine.

This program carries out a steady state phasor analysis of RLC circuits. It is called through the predicate circuit_solve() which has arguments :

- the angular frequency for the analysis.
- the component list.
- the list of nodes which are to be 'grounded' - otherwise all voltages are relative.
- the 'Selection' list - a list of nodes for which computed information is to be printed.

The circuit is defined by a list of components. Each component is described by the component type, name, value and the nodes to which it is connected. The component type is used to determine the component characteristics.

```
circuit_solve(W,L,G,Selection) :-
    get_node_vars(L,NV),
```

```
   write('Returned from get_node_vars'),nl,
   write(L),nl,write(NV),nl,nl,
   c_solve(W,L,NV,Handles,G),
   write('Returned from c_solve'),nl,
   write('W='),write(W),nl,
   write('L='),write(L),nl,
   write('NV='),write(NV),nl,
   write('Handles='),write(Handles),nl,
   write('G='),write(G),nl,!,
   format_print(Handles,Selection).

format_print(Handles, Selection) :-
   write('COMPONENT CONNECTIONS TO NODE '),write(Selection),nl,
   !,write_L(Handles),nl.
write_L([]) :- nl.
write_L([[Component,Name,Value,
   [Node1,c(C1,C2),c(C3,C4)],[Node2,c(C5,C6),c(C7,C8)]]|X1]) :-
   write(Component), write(','), write(Name), write(','),
   writeValue(Component,Value),nl,
   write([Node1,c(C1,C2),c(C3,C4)]),nl,
   write([Node2,c(C5,C6),c(C7,C8)]),
   nl,write_L(X1).
write_L([[Component,Name,Value,
   [Node1,c(C1,C2),c(C3,C4)],[Node2,c(C5,C6),c(C7,C8)],
   [Node3,c(C9,C10),c(C11,C12)]
        ]|X1]) :-
   write(Component),write(','),write(Name),write(','),
   writeValue(Component,Value),nl,
   write([Node1,c(C1,C2),c(C3,C4)]),nl,
   write([Node2,c(C5,C6),c(C7,C8)]),nl,
   write([Node3,c(C9,C10),c(C11,C12)]),nl,
   write_L(X1).
write_L([[Component,Name,Value,
   [Node1,c(C1,C2),c(C3,C4)],[Node2,c(C5,C6),c(C7,C8)],
   [Node3,c(C9,C10),c(C11,C12)],[Node4,c(C13,C14),c(C15,C16)]
        ]|X1]) :-
   write(Component),write(','),write(Name),write(','),
   writeValue(Component,Value),nl,
   write([Node1,c(C1,C2),c(C3,C4)]),nl,
   write([Node2,c(C5,C6),c(C7,C8)]),nl,
   write([Node3,c(C9,C10),c(C11,C12)]),nl,
   write([Node4,c(C13,C14),c(C15,C16)]),nl,
   write_L(X1).
```

```
writeValue(transformer, Value) :-
   write('ratio of '),write(Value),!.
writeValue(resistor, Value) :-
   write(Value),write(' Ohms'),!.
writeValue(diode, Value) :-
   write('type '),write(Value),!.
writeValue(voltage_source, Value) :-
   write(Value),write(' Volts'),!.
writeValue(inductor, Value) :-
   write(Value),write(' Henry'),!.
writeValue(capacitor, Value) :-
   write(Value),write(' Farads'),!.
writeValue(X, Value) :-
   write(Value),write(' unknown').

get_node_vars([],[]).
get_node_vars([[Comp,Num,X,Ns]|Ls],NV) :-
   get_node_vars(Ls,NV1),
   insert_list(Ns,NV1,NV).

insert_list([],NV,NV).
insert_list([N|Ns],NV1,NV3) :-
   insert_list(Ns,NV1,NV2),
   insert(N,NV2,NV3).

insert(N, [], [[N,V,c(0.0,0.0)]]) :- !.
insert(N, [[N,V,I]|NV1], [[N,V,I]|NV1]) :- !.
insert(N, [[N1,V,I]|NV1], [[N1,V,I]|NV2]) :-
   insert(N,NV1,NV2).

c_solve(W,[X|Xs],NV,[H|Hs],G) :-
   addcomp(W,X,NV,NV1,H),
   c_solve(W,Xs,NV1,Hs,G).
c_solve(W,[],NV,[],G) :-
   write('********Checking for zero currents'),nl,
   debug_zero_currents(NV),
   write('********Currents are zero'),nl,
   ground_nodes(NV,G).

debug_zero_currents(NV) :- zero_currents(NV).
debug_zero_currents(NV) :- write('Backtracking'),nl,fail.

zero_currents([[N,V,c(A,B)]|Ls]) :-
   {[[float,[A=0.0,B=0.0]]},zero_currents(Ls).
```

```
zero_currents([]).

ground_nodes(Vs, [N|Ns]) :-
   ground_node(Vs,N), ground_nodes(Vs,Ns).
ground_nodes(Vs,[]).
ground_node([[N,c(0.0,0.0),I]|Vs],N).
ground_node([[N1,V,I]|Vs],N) :- ground_node(Vs,N).

/* Rules to define component charactersitics */
addcomp(W, [Comp2, Num, X, [N1,N2]],NV,NV2,
   [Comp2, Num, X, [N1,V1,I1],[N2,V2,I2]]) :-
   write('Adding '),write(Comp2),nl,
   c_neg(I1,I2),
   iv_reln(Comp2, I1, V, X, W),
   c_add(V,V2,V1),
   subst([N1,V1,Iold1],[N1,V1,Inew1],NV,NV1),
   subst([N2,V2,Iold2],[N2,V2,Inew2],NV1,NV2),
   c_add(I1,Iold1,Inew1),
   c_add(I2,Iold2,Inew2).

/* Specific current/voltage relations for the
   two terminal components */
iv_reln(resistor,I,V,R,W) :-
   c_mult(I,c(R,0.0),V).
iv_reln(voltage_source,I,V,V,W).
iv_reln(isource,I,V,I,W).
iv_reln(capacitor,I,V,C,W) :-
    c_mult(c(0.0,W*C),V,I).
iv_reln(inductor,I,V,L,W) :-
    c_mult(c(0.0,W*L),I,V).
iv_reln(connection,I,c(Const1,Const2),L,W) :-
   {Const1=0.0,Const2=0.0}.
iv_reln(diode,I,V,D,W) :- diode(D,I,V).

/* three rules per diode type */
diode(in914, c(10.0*DV,0.0), c(V,0.0)) :-
   {[float,[V< -100.0,DV=V+100.0]]}.
diode(in914, c(0.0010*V,0.0), c(V,0.0)) :-
   {[float,[V>= -100.0,V<0.60]]}.
diode(in914, c(100.0*DV,0.0), c(V,0.0)) :-
   {[float,[V>=0.60,DV=V-0.60]]}.

addcomp(W, [transistor, Num, X, [N1,N2,N3]], NV, NV3,
   [transistor, Num, X, [N1,V1,I1],
```

```
            [N2,V2,I2],[N3,V3,I3]]) :-
   write('Adding transistor'),nl,
   transistor(X,R,Gain),
   c_add(I1,I3,IT),
   c_neg(I2,IT),
   c_add(Vin,V2,V1),
   c_mult(I1,c(R,0.0),Vin),
   c_mult(I1,c(Gain,0.0),I3),
   subst([N1,V1,Iold1],[N1,V1,Inew1],NV,NV1),
   subst([N2,V2,Iold2],[N2,V2,Inew2],NV1,NV2),
   subst([N3,V3,Iold3],[N3,V3,Inew3],NV2,NV3),
   subst([N4,V4,Iold4],[N4,V4,Inew1],NV3,NV4),
   c_add(I1,Iold1,Inew1),
   c_add(I2,Iold2,Inew2),
   c_add(I3,Iold3,Inew3),
   c_add(I4,Iold4,Inew4).

/* one for each kind of transistor we wish to consider */
transistor(bc108, 1000.0, 100.0).

addcomp(W, [transformer, Num, X, [N1,N2,N3,N4]], NV, NV4,
   [transformer, Num, X, [N1,V1,I1],[N2,V2,I2],
                                 [N3,V3,I3],[N4,V4,I4]]):-
   write('Adding transformer'),nl,
   c_neg(I1,I2),
   c_neg(I3,I4),
   c_add(Vin,V2,V1),
   c_add(Vout,V4,V3),
   c_mult(Vout,c(X,0.0),Vin),
   c_mult(I1,c(X,0.0),I4),
   subst([N1,V1,Iold1],[N1,V1,Inew1],NV,NV1),
   subst([N2,V2,Iold2],[N2,V2,Inew2],NV1,NV2),
   subst([N3,V3,Iold3],[N3,V3,Inew3],NV2,NV3),
   subst([N4,V4,Iold4],[N4,V4,Inew4],NV3,NV4),
   c_add(I1,Iold1,Inew1),
   c_add(I2,Iold2,Inew2),
   c_add(I3,Iold3,Inew3),
   c_add(I4,Iold4,Inew4).

subst([A,C1,C3],Y,[[A,C2,C4]|L1],[Y|L1]) :-
   c_eq(C1,C2), c_eq(C3,C4).
subst(X,Y,[Z|L1],[Z|L2]) :- subst(X,Y,L1,L2).

/* Complex number arithmetic */
```

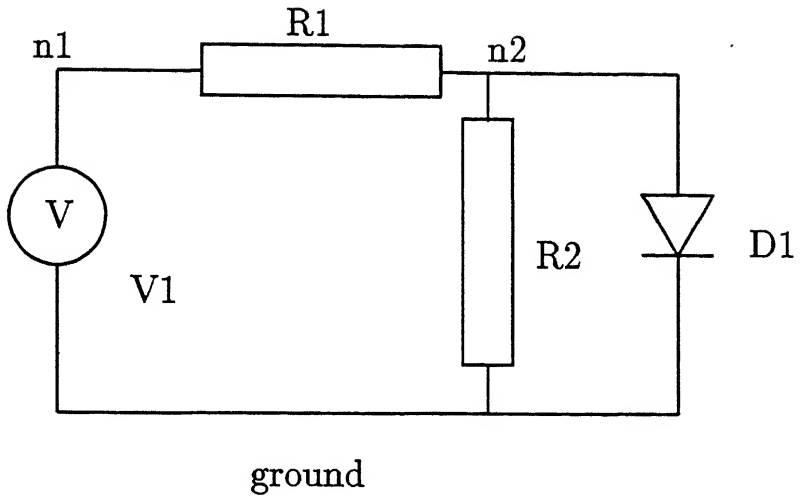


Figure F.1: Use of a general package to solve a circuit

```
c_mult(c(Re1,Im1),c(Re2,Im2),c(NewRe,NewIm)) :-
   {[float,[NewRe=Re1*Re2-Im1*Im2,NewIm=Re1*Im2+Re2*Im1]]}.
c_add(c(Re1,Im1),c(Re2,Im2),c(NewRe,NewIm)) :-
   {[float,[NewRe=Re1+Re2,NewIm=Im1+Im2]]}.
c_neg(c(Re,Im),c(NewRe,NewIm)) :-
   {[float,[NewRe = -Re,NewIm = -Im]]}.
c_eq(c(Re1,Im1),c(Re2,Im2)) :-
   {[float,[Re1=Re2,Im1=Im2]]}.
c_real(c(Re,Im),Re).
c_imag(c(Re,Im),Im).
```

F.7.1 dc_circ1

Consider the problem shown below :

```
goal :-
   {[float,[W=0.0,Vs=10.0,R1=100.0,R2=50.0,Const=0.0]]},
   circuit_solve(W,
      [[voltage_source,v1,c(Vs,Const),[n1,ground]],
       [resistor,r1,R1,[n1,n2]],
       [resistor,r2,R2,[n2,ground]],
       [diode, d1, in914, [n2,ground]]
      ],
      [ground],
      [n2]).
```

The solution obtained is :

```
voltage_source,v1,c(10.0,0.0) Volts
```

```
[n1,c(10.0,0.0),c(-0.0939918,0.0)]
[ground,c(6.66134e-16,0.0),c(0.0939918,0.0)]
resistor,r1,100.0 Ohms
[n1,c(10.0,0.0),c(0.0939918,0.0)]
[n2,c(0.60082,0.0),c(-0.0939918,0.0)]
resistor,r2,50.0 Ohms
[n2,c(0.60082,0.0),c(0.0120164,0.0)]
[ground,c(0.0,0.0),c(-0.0120164,0.0)]
diode,d1,type in914
[n2,c(0.60082,0.0),c(0.0819754,0.0)]
[ground,c(0.0,0.0),c(-0.0819754,0.0)]
```

F.7.2 dc_circ2

```
goal :-
   {[float,[W=100.0,Vs=10.0,
    Tr1=5.0,Tr2=0.2,
    R1=200.0,R2=1000.0,R3=50.0,R4=30.0,
    C1=0.05,L1=0.005,Const=0.0]]
   },
   circuit_solve(W,
      [[voltage_source, v1, c(Vs,Const), [in,ground1]],
       [resistor, r1, R1, [in,n1]],
       [transformer, t1, Tr1, [n1,ground1, n2, ground2]],
       [resistor, r2, R2, [n2,n3]],
       [capacitor, c1, C1, [n3, n4]],
       [resistor, r3, R3, [n4, ground2]],
       [transformer, t2, Tr2, [n4, ground2, out, ground3]],
       [resistor, r4, R4, [out, ground3]],
       [inductor, l1, L1, [out, ground3]]
      ],
      [ground1, ground2, ground3],
      [out]).
```

The solution obtained is :

```
voltage_source,v1,c(10.0,0.0) Volts
[in,c(10.0,1.35525e-20),c(-0.000396825,-7.08639e-08)]
[ground1,c(0.0,1.35525e-20),c(0.000396825,7.08639e-08)]
resistor,r1,200.0 Ohms
[in,c(10.0,1.35525e-20),c(0.000396825,7.08639e-08)]
[n1,c(9.92063,-1.41728e-05),c(-0.000396825,-7.08639e-08)]
transformer,t1,ratio of 5.0
[n1,c(9.92063,-1.41728e-05),c(0.000396825,7.08639e-08)]
[ground1,c(0.0,0.0),c(-0.000396825,-7.08639e-08)]
```

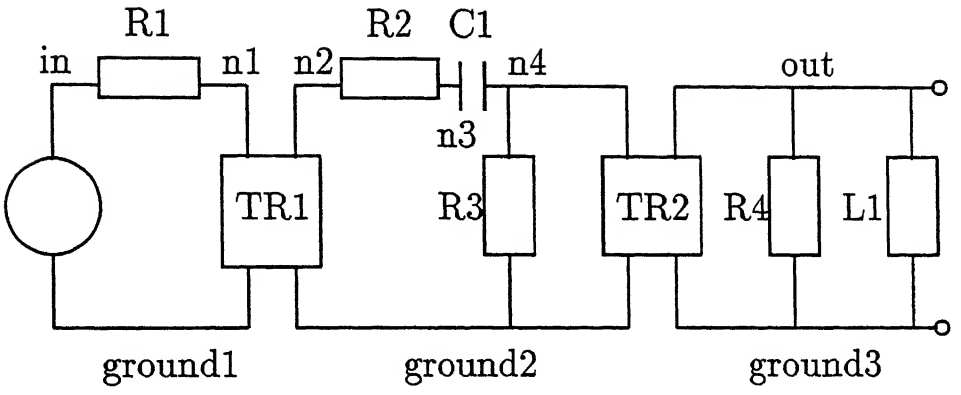


Figure F.2: RLC circuit

```
[n2,c(1.98413,-2.83456e-06),c(-0.00198413,-3.5432e-07)]
[ground2,c(0.0,-3.52366e-18),c(0.00198413,3.5432e-07)]
resistor,r2,1000.0 Ohms
[n2,c(1.98413,-2.83456e-06),c(0.00198413,3.5432e-07)]
[n3,c(7.40831e-07,-0.000357154),c(-0.00198413,-3.5432e-07)]
capacitor,c1,0.05 Farads
[n3,c(7.40831e-07,-0.000357154),c(0.00198413,3.5432e-07)]
[n4,c(6.69967e-07,3.96711e-05),c(-0.00198413,-3.5432e-07)]
resistor,r3,50.0 Ohms
[n4,c(6.69967e-07,3.96711e-05),c(1.33993e-08,7.93422e-07)]
[ground2,c(-1.73536e-19,0.0),c(-1.33993e-08,-7.93422e-07)]
transformer,t2,ratio of 0.2
[n4,c(6.69967e-07,3.96711e-05),c(0.00198411,-4.39102e-07)]
[ground2,c(0.0,0.0),c(-0.00198411,4.39102e-07)]
[out,c(3.34983e-06,0.000198355),c(-0.000396823,8.78204e-08)]
[ground3,c(8.67362e-19,0.0),c(0.000396823,-8.78204e-08)]
resistor,r4,30.0 Ohms
[out,c(3.34983e-06,0.000198355),c(1.11661e-07,6.61185e-06)]
[ground3,c(0.0,0.0),c(-1.11661e-07,-6.61185e-06)]
inductor,l1,0.005 Henry
[out,c(3.34983e-06,0.000198355),c(0.000396711,-6.69967e-06)]
[ground3,c(0.0,0.0),c(-0.000396711,6.69967e-06)]
```

F.8 Dirichlet Problem

The following problem solves the Dirichlet problem for Laplace's equation in two-dimensions. The program outputs a matrix (list of lists) giving the temperature of a surface at discrete points. Typical input is a matrix which contains specific values at the four boundaries. The program then specifies that the temperature at each non-boundary point is the average of those of four neighbouring points.

```
laplace([H1,H2,H3|T]) :-
   av(H1,H2,H3),
   laplace([H2,H3|T]).
laplace([_,_]).

av([TL,T,TR|T1],[ML,M,MR|T2],[BL,B,BR|T3]) :-
   {B+T+ML+MR-4.0*M=0.0},
   av([T,TR|T1],[M,MR|T2],[B,BR|T3]).
av([_,_],[_,_],[_,_]) :-
   write('last av'),nl.

goal1 :-  /* dirichlet1 */
   A=[
   [0.0,   0.0,   0.0,   0.0,   0.0],
   [100.0, R,     S,     T,     100.0],
   [100.0, U,     V,     W,     100.0],
   [100.0, X,     Y,     Z,     100.0],
   [100.0, 100.0, 100.0, 100.0, 100.0]
     ],
   laplace(A),
   write(A),nl,nl.

Result : [[0.0,   0.0,     0.0,     0.0,     0.0],
          [100.0, 57.1429, 47.3215, 57.1426, 100.0],
          [100.0, 81.2501, 75.0005, 81.2485, 100.0],
          [100.0, 92.8571, 90.1787, 92.8574, 100.0],
          [100.0, 100.0,   100.0,   100.0,   100.0]
         ]

goal2 :-  /* dirichlet2 */
   A=[
   [0.0,   0.0,   0.0],
   [100.0, X,     100.0],
   [100.0, B,     100.0]
     ],
   laplace(A),
   write(A),nl,nl.

Result : [[  0.0,                    0.0,   0.0],
          [100.0, +(50.0, *(0.25, _B)), 100.0],
          [100.0,                  _B, 100.0]
         ]
```

Appendix G

Scanner and Parser for Compiling Clauses

G.1 Scanner

```
/* Terminals given by scanner in LEX */
%right _atom
%right _qdash _if
%right _semicolon
%right _comma
%right _is _univ _equal _not_equal
%left _less_than _equal_to_less_than _greater_than_equal_to
      _greater_than
%right _eequal _not_eequal
%left _minus _plus
%left _idiv _slash _asterix _mod
%right _not
%right UNARY_minus UNARY_plus
%token _char _var _real _int _str _error
%token _period _tail_sym
%token _open_round_bracket _close_round_bracket
%token _open_square_bracket _close_square_bracket
%token _open_curly_bracket _close_curly_bracket
```

G.2 Parser

```
/* Grammer for Parser in YACC */
PGM:
        |
        PGM CLAUSE
        ;
```

```
CLAUSE:
          _qdash _open_square_bracket _atom
          _close_square_bracket _period
        | _if _open_square_bracket _atom
          _close_square_bracket _period
        | EXPRESSION TAIL _period
        ;
EXPRESSION:
        LITERAL
        | EXPRESSION _atom EXPRESSION
        | EXPRESSION _is EXPRESSION
        | EXPRESSION _univ EXPRESSION
        | EXPRESSION _equal EXPRESSION
        | EXPRESSION _not_equal EXPRESSION
        | EXPRESSION _less_than EXPRESSION
        | EXPRESSION _equal_to_less_than EXPRESSION
        | EXPRESSION _greater_than_equal_to EXPRESSION
        | EXPRESSION _greater_than EXPRESSION
        | EXPRESSION _eequal EXPRESSION
        | EXPRESSION _not_eequal EXPRESSION
        | EXPRESSION _minus EXPRESSION
        | EXPRESSION _plus EXPRESSION
        | EXPRESSION _idiv EXPRESSION
        | EXPRESSION _slash EXPRESSION
        | EXPRESSION _asterix EXPRESSION
        | EXPRESSION _mod EXPRESSION
        | _plus EXPRESSION                    %prec UNARY_plus
        | _minus EXPRESSION                    %prec UNARY_minus
        | _not        EXPRESSION
        | _if _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _semicolon _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _comma _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _is _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _univ _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _equal _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _not_equal _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _less_than _open_round_bracket EXPRESSION _comma
```

```
          EXPRESSION _close_round_bracket
        | _equal_to_less_than
          _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _greater_than_equal_to
          _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _greater_than
          _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _eequal _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _not_eequal _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _minus _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _plus _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _idiv _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _slash _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _asterix _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        | _mod _open_round_bracket EXPRESSION _comma
          EXPRESSION _close_round_bracket
        ;
LITERAL:
          _int
        | _real
        | _char
        | _str
        | _open_curly_bracket COMMAD_EXPRESSION
          _close_curly_bracket
        | _open_round_bracket EXPRESSION _if
          COMMAD_EXPRESSION _close_round_bracket
        | _open_round_bracket EXPRESSION _semicolon
          EXPRESSION _close_round_bracket
        | _open_round_bracket COMMAD_EXPRESSION
          _close_round_bracket
        | VARSYM
        | _atom PROCESSATOM
        | LISTLITERAL ;
COMMAD_EXPRESSION:
```

```
        COMMAD_EXPRESSION _comma COMMAD_EXPRESSION
        | EXPRESSION ;
PROCESSATOM: PARLIST ;
PARLIST:_open_round_bracket ARGLIST _close_round_bracket
        | ;
ARGLIST:TERM ARGTAIL ;
ARGTAIL:_comma ARGLIST
        | ;
TERM:         EXPRESSION        ;
VARSYM:         _var ;
TAIL:         _if LITLIST | ;
LITLIST:EXPRESSION LITTAIL ;
LITTAIL:_comma LITLIST | ;
LISTLITERAL:
          _open_square_bracket LISTELEMENTS
          _close_square_bracket ;
NONNULLLISTELEMENTS:
          TERM LISTELEMENTSLIST ;
NULLLISTELEMENTS: ;
LISTELEMENTS:
          NONNULLLISTELEMENTS
        | NULLLISTELEMENTS ;
LISTELEMENTSLIST:
          _comma NONNULLLISTELEMENTS
        | _tail_sym LISTCDR
        | NULLLISTELEMENTS ;
LISTCDR:VARSYM
        | LISTLITERAL ;
```